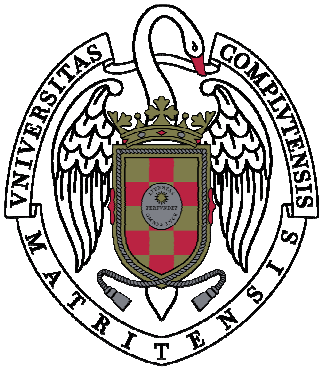
Proyecto de Sistemas Informáticos



Universidad Complutense de Madrid

Facultad de informática

Curso 2009-2010.

M.A.G.

Massive Avatar Generation

Herramientas para la visualización, personalización y producción masiva de personajes tridimensionales.

Profesores directores:	**Pablo Gervás Gómez-Navarro** **Gonzalo Rubén Méndez Pozo**
Autores:	**José María Méndez González** **Lidia Martínez Prado** **Gabriel López de Armas**

<u>Cesión de derechos:</u>

Los abajo firmantes, Gabriel López de Armas, José María Méndez González y Lidia Martínez Prado, autores del proyecto MAG (Massive Avatar Generation) en la asignatura de Sistemas Informáticos, autorizan a la Universidad Complutense de Madrid a difundir y utilizar los contenidos de este proyecto, así como el código, prototipos o documentación asociada a dicho proyecto, con fines exclusivamente académicos, nunca comerciales, y mencionando expresamente a sus autores.

Participantes:

Gabriel López de Armas **José María Méndez González** **Lidia Martínez Prado**

CONTENIDO

1 - RESUMEN

Imagen 1: Diferentes cangrejos generados con MAG (renderizados con MentalRay©).

MAG (Generación Masiva de Avatares) es un conjunto de herramientas para la visualización y personalización de personajes tridimensionales. Las herramientas existentes actualmente para realizar esta tarea son restrictivas en cuanto al tipo de personajes que permiten generar, y la visualización interactiva de los mismos es, por lo general, pobre.

Solucionamos estos problemas mediante un formato propio de almacenamiento de recursos gráficos basado en el estándar Collada (paquetes .mag) ; un API escrito en C++ que permite la empaquetación de recursos gráficos y su posterior manipulación (librerías MAGInput y MAGOutput) ; y una interfaz de edición y visualización avanzada de personajes (editor MAGTools).

Gracias a esto se pueden producir grandes cantidades de personajes distintos en poco tiempo, y debido a su genericidad, no hay restricción alguna en cuanto al tipo de personajes generables.

PALABRAS CLAVE

3D, gráficos, gráficos tridimensionales, personajes, renderizado, generación masiva, visualización, interpolación.

ABSTRACT

MAG (Massive Avatar Generation) is a tridimensional character visualization and customization toolkit. The currently available tools for this task are restrictive in the kind of characters generable by the user, and the visualization provided by them is generally poor.

We solve these issues by the means of our own graphic resources container format built upon the Collada industry standard (.mag package), an easy-to-use C++ API which allows us to pack and manipulate graphic resources (MAGInput and MAGOutput libraries), and an advanced edition and visualization interface (MAGTools editor).Thanks to our system, it is possible to generate massive amounts of unique characters in less time than with traditional methods, and because of its genericity, there´s no restriction at all in terms of kinds of generable characters.

2 - INTRODUCCIÓN

Explicaremos, para empezar, la motivación del proyecto: las razones por las que creemos que es necesario facilitar las tareas a los artistas de gráficos y puedan dar rienda suelta a su creatividad sin restricciones y por qué es importante.

Después explicaremos el propósito del proyecto: en qué tareas proporcionará ayuda adicional a todas esas carencias que expresamos en la motivación. Para ello mostramos un ejemplo práctico de caso de uso del conjunto de herramientas desarrolladas, y una lista describiendo brevemente los elementos que componen el proyecto.

Luego explicamos la viabilidad del proyecto: exponemos nuestra investigación inicial y algunos puntos que tuvimos que estudiar para que los procesadores actuales pudieran soportar el proyecto.

2.1 - MOTIVACIÓN

En el mercado del entretenimiento y el ocio cada vez tienen más peso los videojuegos y aplicaciones interactivas tridimensionales. Cada día se reducen más las diferencias entre un videojuego y una película, tanto en el aspecto visual como en el narrativo. Una gran parte de su realismo y credibilidad reside en la calidad de los personajes o avatares gráficos. Su importancia es tal que, si carecen de la calidad y variedad suficiente, se pierde la ilusión de inmersión en el mundo representado.

Consideraremos un **avatar** como una figura tridimensional compuesto por mallas (una serie de polígonos colocados en el espacio), texturas (imágenes dibujadas en la superficie de las mallas), un esqueleto (cuyos “huesos” tendrán relaciones con las mallas, para poder animarlas) y animaciones (secuencias de movimiento de estos huesos).

Para conseguir esta meta, en el equipo de desarrollo de una aplicación que represente un mundo virtual han de hacer coincidir el aspecto técnico y el artístico de manera que ambos sean complementarios y el uno no limite o fuerce al otro, algo que suele suceder con frecuencia: los artistas encargados de llevar a cabo la realización de los personajes (**grafistas o artistas gráficos** de ahora en adelante) suelen ver su libertad creativa coartada bien por las necesidades técnicas del motor gráfico[1], por la plataforma para la que se está desarrollando o bien por el límite de tiempo del que disponen para llevar a cabo su trabajo.

Además el proceso de creación de personajes suele ser lento y tedioso ya que requiere una gran dosis de visión artística que no puede ser automatizada de ninguna manera. El tiempo requerido en llevar esta tarea a cabo compensa cuando se requiere un personaje con unas características muy específicas o que representa una parte importante para el resultado final, como puede ser un personaje protagonista, pero también es necesario detallar lo suficiente el resto de avatares que

[1] Motor gráfico: Software utilizado para la visualización de gráficos en un computador.

aparezcan en la aplicación pese a que su aparición sea breve, ya que su contribución a la ilusión de inmersión puede ser igual o incluso más importante.

Imagen 1 : Crysis, de Electronic Arts ©: Un ejemplo de realismo gráfico

Por si esto fuera poco, en muchos casos los grafistas se ven a sí mismos llevando a cabo una y otra vez el proceso de creación de las mismas estructuras anatómicas básicas de una criatura bípeda o cuadrúpeda, o modificando una base anteriormente realizada de acuerdo con el resultado que necesitan obtener. Lo mismo se aplica para todo tipo de objetos y complementos para los personajes: sombreros, cascos, pulseras, espadas, collares, etc.

Una vez han conseguido el resultado final queda la integración con el motor gráfico elegido, proceso que no está estandarizado ni automatizado de ninguna manera, y que a veces exige ajustes en los avatares de forma individual para su correcta exportación desde el paquete de animación/modelado elegido a la aplicación. Incluso en algunos casos la elección tanto del programa/paquete de diseño que han de emplear los grafistas como del motor gráfico se lleva a cabo teniendo en cuenta el método de exportación que se piensa seguir, limitando una vez más las posibilidades de grafistas (y de programadores) al verse obligados a trabajar con una herramienta específica con la que es posible que no estén familiarizados a priori.

MAG nace con la idea de linealizar, simplificar y acelerar lo más posible el proceso de creación de personajes, reutilización de recursos gráficos, y su integración en los distintos entornos y motores gráficos empleados en la industria del entretenimiento digital, permitiendo la generación de cantidades masivas de avatares diferenciados entre sí en una fracción del tiempo que tradicionalmente se requiere para crear uno solo. De esta manera muchas de las limitaciones existentes en el campo de la creación de personajes disminuyen o simplemente desaparecen. Esto repercute directamente en el tiempo de desarrollo de cualquier aplicación que incluya avatares gráficos, permitiendo emplear el tiempo ahorrado en su creación en mejorar otros aspectos de la aplicación o simplemente acelerar su lanzamiento al mercado.

2.2 – PROPÓSITO DEL PROYECTO

El objetivo del proyecto, explicado de forma concisa, es proporcionar una herramienta en forma de librerías para poder generar personajes tridimensionales diferenciados. El desarrollador de una aplicación que use gráficos tridimensionales se puede beneficiar de la herramienta para poder mostrar en pantalla distintos personajes del mismo tipo (con morfología parecida) pero con diferencias entre sí (por ejemplo, altura diferente).

Éstos personajes se crearán a partir de permutaciones de los propios recursos que un supuesto diseñador ya ha creado e introducido en el formato .mag, creado para el proyecto.

Para explicarlo mejor se narra un ejemplo de uso del proyecto:

- Un diseñador decide crear un recurso .mag que permita generar estudiantes de una facultad. Para ello, usando un software de terceros creará distintos personajes humanos, incluyendo texturas y animaciones. Cuando haya finalizado, y usando la librería de entrada del proyecto, compondrá un paquete .mag con todos los recursos que ha creado.
- El desarrollador de una aplicación decide que ésta requiere mostrar tridimensionalmente una facultad, y además piensa que es buena idea introducir a estudiantes para dar más realismo a la escena. Para ello, adquiere del desarrollador el paquete .mag anteriormente mencionado, y usando la librería de salida del proyecto, la carga dentro de su aplicación. A partir de ese momento, podrá obtener el número que desee de personajes, que serán estudiantes, y todos ellos serán diferentes a los anteriores. El grado de diferencia ha sido impuesto por el diseñador, y dependiendo de éste puede que los cambios sean muy sutiles (diferencias de altura o complexión) o notables (cambios de color de piel y vestimenta, morfología humana diferenciada).
- El desarrollador de la aplicación podrá mostrar una facultad llena de estudiantes diferenciados entre sí, que proporcionará más realismo a la escena.

Por ello, el conjunto del proyecto contiene las siguientes piezas:

- **MAGInput** - Una librería C++ de creación de paquetes de recursos. Idealmente será usada por otras aplicaciones diferentes a nuestro editor. Esta librería tomará los recursos (mallas, texturas, animaciones, esqueletos) y metadatos, generando con ellos un paquete .mag.
- **MAGOutput** - Una librería C++ de carga de paquetes de recursos. Esta permitirá emplear los recursos de un paquete para generar personajes.
- **MAGTools** - Un editor gráfico para poder controlar de forma cómoda las funcionalidades de las dos librerías anteriores y la exportación de uno o multitud de avatares. Constituye el centro del proyecto, pues reúne todos los módulos de éste.
- Ésta **memoria** como referencia para poder utilizar las herramientas.
- **Varios paquetes** para demostrar que el proyecto funciona adecuadamente, y servir como ejemplo de uso y aprendizaje de la nueva herramienta.

2.3 – ESTUDIO PREVIO DE VIABILIDAD DEL PROYECTO.

Antes de comenzar el desarrollo del proyecto dudamos de la posibilidad de llevar a cabo la funcionalidad requerida empleando únicamente algoritmos sencillos y eficientes, como es la interpolación lineal o esférica[2] para mezclar los elementos y obtener las variaciones resultantes de esta mezcla. Tras documentarnos debidamente, se hizo evidente que no había ningún tipo de restricción teórica o tecnológica y que no iba a ser necesario emplear métodos excesivamente complejos. Aun así, hemos tenido en cuenta el empleo de algunos algoritmos de renderizado o de interpolación algo más avanzados, poniendo como ejemplo una técnica más avanzada propuesta por la empresa *Valve* que mencionamos en el apartado 6.3.4.

Se estableció la necesidad de dar soporte a datos y efectos gráficos empleados por motores gráficos de última generación. Estos motores tratan de ofrecer el mayor realismo visual posible, como son el CryEngine2[3] o el Unreal Engine 3[4], siendo los dos software muy avanzados en este campo y de uso extendido en el mercado. Durante el estudio de los mismos se escogió el conjunto de información soportada de forma que los avatares generados luciesen lo mejor posible en la mayoría de ellos, tomando CryEngine2 como referencia y meta en cuanto a calidad gráfica de los avatares generados.

Se ignoraba el límite de la capacidad de los sistemas actuales para operar con gran cantidad de matrices (operación principal en aplicaciones orientadas a gráficos) y de realizar procesado complejo de elementos gráficos. Para ello se realizó una pequeña aplicación de prueba donde se mezclaban por interpolación de forma secuencial y de forma paralela grandes cantidades recursos gráficos (más tarde sólo sería de utilidad el mezclado secuencial, como veremos en la sección 6) y donde se realizaba una sencilla animación mediante esqueleto[5] de un personaje. Se determinó que el límite de huesos por esqueleto implementados en hardware estaba en 256 para una tarjeta gráfica de gama media, pero que exceptuando esa limitación, todo lo propuesto inicialmente era viable.

---

[2] Ver apartado 6.3.
[3] Página oficial de Crytek, desarrolladora de CryEngine: http://www.crytek.com/
[4] Página oficial de la tecnología Unreal: http://www.unrealtechnology.com/
[5] Ver Apartado 3: El proceso de creación de un personaje.

3 – ESTADO DEL ARTE

En este apartado explicaremos la problemática sobre el tipo de proyecto que hemos realizado y algunos ejemplos de desarrollos de última tecnología realizados que hemos encontrado que han sido probados en la industria y han sido acogidos y aceptados por la comunidad de grafistas y empresas en general.

3.1 - INTRODUCCIÓN

El problema que estamos tratando, la generación automática de avatares 3D a partir de un conjunto de recursos gráficos, es un tema que ya ha sido tratado ampliamente. Equipos de desarrollo de videojuegos, de efectos especiales o simples proyectos de investigación han llegado de forma independiente a diversas soluciones.

El enfoque tradicional para resolver el problema es la fuerza bruta y el trabajo artesanal, ya que generalmente recompensa con una mayor variedad de resultados y alta calidad de los mismos: los artistas producen a mano todos los personajes necesarios, pero esto tiene un alto coste ya que, aunque se reutilicen recursos, cada personaje puede llevar varias horas o incluso días de trabajo.

Por otro lado, este enfoque no siempre es posible de llevar a cabo, ya que a veces la posibilidad de crear avatares desde dentro de la propia aplicación es un requisito del sistema (algo habitual en videojuegos de rol, shoot´em ups o de simulación, o el ejemplo citado en la imagen 2). Por ello, la solución comúnmente adoptada pasa por realizar una interfaz de usuario desde la cual se puedan ajustar diversos parámetros de un personaje o personajes predeterminados, pudiendo modificar constitución, color de piel, ojos, pelo, etc. Esta es una forma sencilla de resolver el problema e intuitiva para el usuario, pero está limitada por el propio diseño de la interfaz y por el tipo y grado de personalización permitida en los avatares.

Imagen 2: Ejemplo de creador de personajes en Sims3©.

MAG tiene muchos puntos en común con este tipo de editores integrados en aplicaciones específicas, pero busca abstraer los parámetros de generación de avatares, haciendo el proceso de edición independiente del aspecto de éstos, y del nivel de personalización que admiten, y poder emplearlos en cualquier aplicación gráfica que cumpla con los requisitos técnicos necesarios.

3.2 – REVISIÓN DE SOLUCIONES SIMILARES

Tras llevar a cabo una labor de investigación, tratando de encontrar soluciones similares a la nuestra, se han encontrado dos productos comerciales que comparten ciertas características con nuestro proyecto: Quidam de N-Sided y Poser de Smith Micro Software.

Ambos permiten la creación de variaciones de personajes de forma intuitiva, y pueden exportar el personaje creado a distintos formatos, entre ellos Collada. Ninguno de los dos productos es gratuito, costando el primero 200$ y el segundo 250$ en sus versiones básicas.

En una lectura inicial, entre las características anunciadas de ambos productos se cuentan la creación sencilla en intuitiva de personajes y la personalización de diferentes aspectos del mismo, abstrayéndose del tipo de avatar con que se esté trabajado.

Los dos productos cuentan con versiones de prueba limitadas. En ningún caso se pretende realizar una investigación rigurosa ni un análisis de las aplicaciones, solo se pretende descubrir si ya llevan a cabo lo que pretende conseguir MAG.

3.2.1 - POSER

Imagen 3: de las pantallas del software Poser

Poser[6], ilustrado en la imagen 3, es un software especialmente potente, con multitud de opciones. Cuenta con una comunidad de usuarios bastante madura, y existen diversos plugins para aumentar su funcionalidad. Así mismo, su funcionamiento es parecido al de Quidam. Dados unos "packs", el usuario puede utilizarlos como base para generar nuevos avatares y modificar diversos aspectos, como la ropa, el pelo, etc.

Permite además cargar personajes, luces y cámaras para fotos y videos animados.

Utiliza formato OBJ, un sencillo formato comúnmente utilizado que guarda los datos de todos los elementos que toman parte en la escena. El software aporta una librería completa de humanos, animales, robots y figuras "cartoon". También incluye algunas poses, distintos tipos de pelos, texturas, gestos y expresiones faciales.

Los comentarios que se pueden ver sobre el software, que son los que más nos interesan para ver qué pide la comunidad, son que el programa ha quedado un poco fuera de lugar porque no incorpora las características de los potentes nuevos softwares de 3D, que debes crear los modelos utilizando otra herramienta.

Las ventajas es que lo que genera puede ser de nuevo introducido en software como Maya, Carrara y 3DsMax.

A día de hoy, la última actualización data de Marzo de 2010, y ha crecido gracias a la comunidad de usuarios.

3.2.2 - QUIDAM 3D

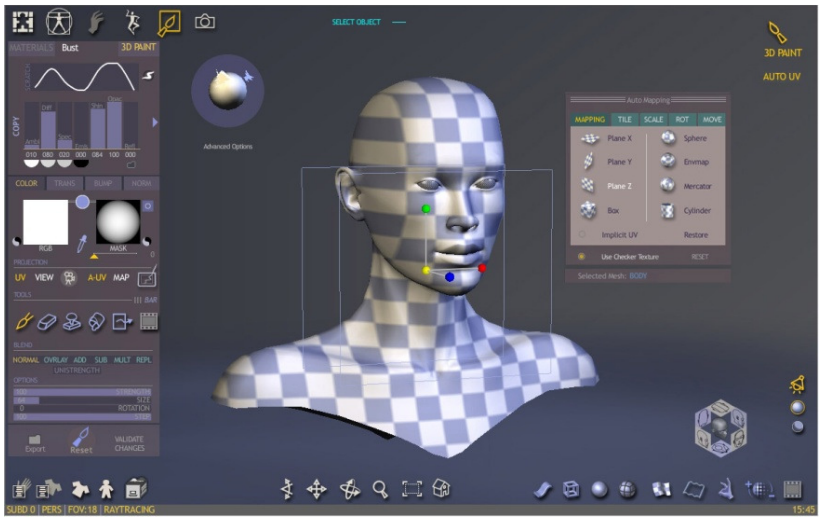


Imagen 4: Una de las pantallas de edición de Quidam 3D

[6] Página web oficial de Poser: http://poser.smithmicro.com/

En el caso de Quidam[7], ilustrada en la imagen 4, es una aplicación muy completa y bastante intuitiva, permitiendo configurar casi cualquier característica de un personaje, desde su fisonomía hasta las texturas y animaciones. Todo ello se plantea de un modo totalmente gráfico, con bastantes opciones por defecto, evitando así que el usuario necesite incluir software adicional en su flujo de trabajo. Esta es una gran diferencia respecto a Poser.

Sin embargo, aunque las capacidades de edición de un modelo son muy abundantes, no es posible para el usuario llevar a cabo la creación o edición de un modelo que no haya sido producido por la empresa. Es decir, están disponibles comercialmente una serie de "Model packs", algunos incluidos con el software y otros a la venta de forma separada, y cada uno de ellos permite la edición de un tipo de personaje: humanos, perros, personajes cartoon, etc, pero todos con una serie de opciones limitadas. Por ejemplo, empleando este software no se puede conseguir un modelo de un cangrejo, pues la empresa no ha realizado un "Model pack" de cangrejos y no es posible para el usuario realizar un paquete de cangrejos si así lo deseara.

Hemos descubierto bastantes quejas referentes a este problema, y al precio que imponen en la utilización de nuevos objetos 3d. Un gran problema que también suele comentarse es que los modelos de personajes tienen serios problemas en sus proporciones físicas, y que su aspecto es bastante poco estético.

No hay mucha más información sobre este programa, con lo que podemos ver que algunas de sus características y la poca libertad que permite hacen que no exista una importante comunidad de usuarios en torno a él.

3.3 - CONCLUSIÓN

Como ya se ha comentado, el problema de la generación "automática" de avatares ha sido solucionado de diversas formas, por lo general acotando el tipo de personajes que se pueden generar a cambio de ofrecer al usuario más comodidad y facilidad para editar pequeños detalles. Este es el caso de Poser y Quidam, donde es necesario disponer de unos determinados paquetes de recursos para poder realizar un personaje.

La tecnología de MAG es similar en cuanto a filosofía y arquitectura, pero intenta centrarse más en el mercado profesional ofreciendo mayor libertad para poder generar personajes, ya que si se dispone de los recursos gráficos, es posible diseñar y crear paquetes propios que funcionan perfectamente en el editor sin ningún tipo de requisito adicional. Con MAG también se puede llevar a cabo la misma estrategia de mercado empleada por los dos programas que hemos analizado, es decir, se pueden vender paquetes de recursos realizados por un grupo de artistas especializados. Pero lo esencial, es que la creación de un paquete de recursos debe resultar fácil, con herramientas abiertas y que cualquier usuario pueda hacerlo (con mejores o peores resultados dependiendo de su habilidad), no únicamente la empresa propietaria del software.

[7] Página oficial de N-Sided, desarrollador de Quidam: http://www.n-sided.com/

4 - EL PROCESO DE CREACIÓN DE UN PERSONAJE (ASPECTO ARTÍSTICO)

En esta sección se explicarán los pasos necesarios para realizar un personaje tridimensional completo, de manera que el lector pueda apreciar la complejidad del proceso cuando es llevado a cabo “a mano” (la forma habitual) y así comprender mejor el propósito del proyecto. Por lo general se puede resumir en 4 etapas básicas, aunque dependiendo de las necesidades propias de cada desarrollo pueden incluirse etapas nuevas o prescindir de algunas:

- Modelado.
- Desplegado de coordenadas de textura y realización de texturas.
- Esqueleto, Skinning y Setup.
- Animación.

Cada una de estas etapas se detallará en las secciones siguientes.

4.1 - MODELADO

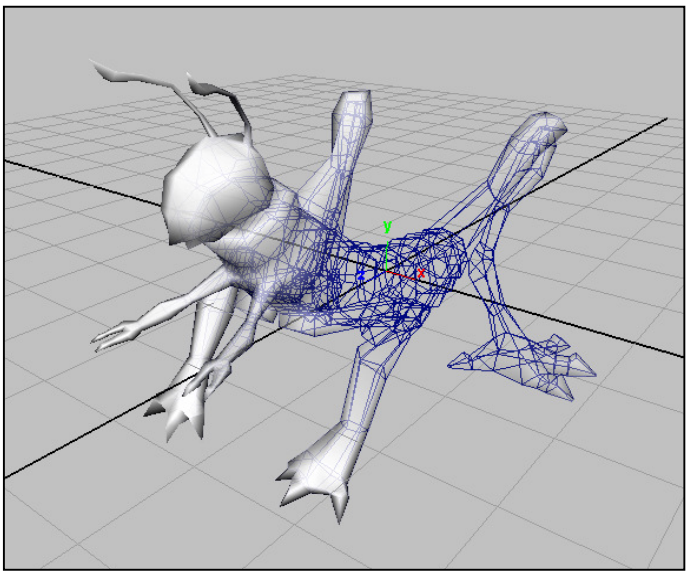


Imagen 5: Un modelo y su wireframe (rejilla geométrica estructural)

La primera etapa consiste en el paso más importante, dar forma al objeto que queremos representar. Para ello, existen diversos paradigmas de representación digital de objetos y superficies en un entorno tridimensional, (vóxeles, superficies paramétricas, puntos, etc) siendo el más extendido la representación mediante superficies poligonales, que a partir de ahora llamaremos **mallas**.

Los componentes básicos de cualquier objeto representado mediante este paradigma son, pues, los **vértices**: una colección de puntos en el espacio definidos mediante tres coordenadas x,y,z. Estos vértices están interconectados para formar **polígonos** de n lados, aunque los más comunes son los triángulos y los polígonos de cuatro lados, denominados coloquialmente “quads”.

Adicionalmente, cada vértice puede contener información extra: un vector tridimensional denominado **normal** del vértice, otro vector tridimensional ortogonal a la normal denominado **tangente**, una o varias coordenadas bidimensionales llamadas **coordenadas de textura**, una o varias parejas <entero, numero en punto flotante> denominadas **pesos de animación**, una terna de valores enteros entre 0 y 255 denominada **color** del vértice, y otros datos según necesidad, muchos de ellos destinados a calcular efectos de iluminación sobre la superficie del polígono.

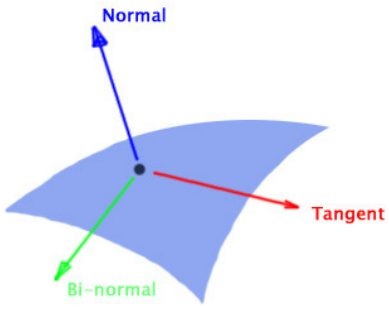


Imagen 6 : Los tres vectores de un vértice

Para cada vértice de una malla compuesta de vértices, los vectores normal, tangente y bitangente o binormal forman la base de un espacio vectorial denominado “espacio de tangente”, que suele ser empleado para realizar en él cálculos de iluminación de la superficie. La matemática necesaria para obtener la tangente y bitangente dado el triángulo al que pertenece el vértice, su normal, y las coordenadas de textura de los vértices del triángulo es relativamente sencilla, y no la detallaremos aquí. Si se dispone desde el principio de al menos dos de los tres vectores, el tercero puede ser hallado mediante un simple producto vectorial entre ambos.

Las normales y las coordenadas de textura suelen interpolarse a lo largo de la superficie de los polígonos para disimular la naturaleza discreta de este tipo de representación (en ello se basa el método de sombreado Goraud [GOR1971], por ejemplo). Otros métodos están basados en renderizado de puntos, o en superficies paramétricas (superficies **NURBS**[8]) que en principio ofrecen una representación continua carente de vértices o aristas, pero que suelen subdividirse (o “teselarse”) para conseguir una representación poligonal.

En la siguiente página tenemos un ejemplo comparativo entre unas superficie poligonal (abajo y a la derecha) y una superficie paramétrica Nurbs (arriba a la izquierda). A pesar de que las nurbs se evalúan y se convierten a polígonos a la hora de renderizar usando hardware y lo único que cambia es la manera de definir la superficie (ecuaciones paramétricas en lugar de una lista de puntos) y no las primitivas que la componen, en la práctica, sobre todo en lo que a renderizado en tiempo real se refiere, es mucho más común el empleo de polígonos directamente.

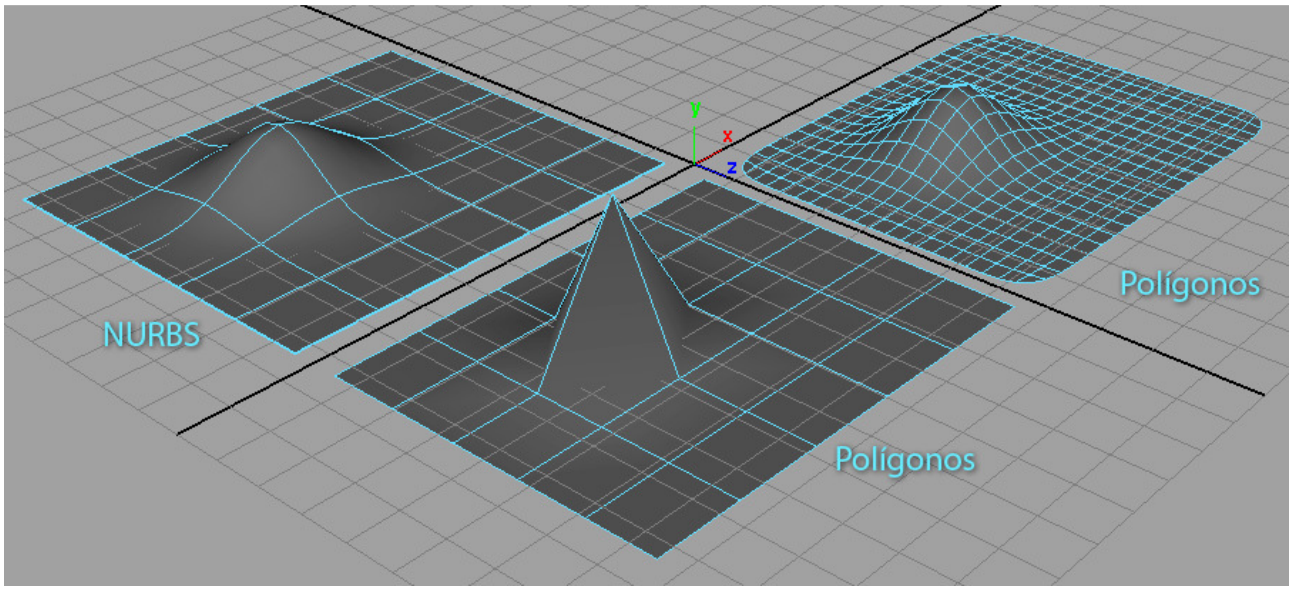


Imagen 7 : Variedades de mallas geométricas

Mediante la manipulación de primitivas geométricas básicas empleando diversas herramientas -extrusión de caras poligonales, desplazamiento de vértices, creación de aristas, por nombrar algunas- es posible crear una superficie poligonal que aproxime tanto como queramos una forma

[8] NURBS: Non Uniform Rational B-Splines. Desarolladas a partir del trabajo de Pierre Étienne Bézier para Renault.

determinada. De esta manera se crea prácticamente la totalidad del contenido artístico de los videojuegos tridimensionales excepto algunos efectos concretos que están empezando a ser representados de forma volumétrica mediante vóxeles[9] y otras técnicas similares en aplicaciones modernas.

Una vez se posee la **malla** geométrica que representa la superficie del personaje y sus complementos (vestuario, objetos) es posible (y necesario la mayor parte de las veces) aumentar el detalle que podemos alcanzar empleando solamente vértices coloreados. Para eso se utilizan las texturas, explicadas en la siguiente sección.

Para más información sobre principios de representación 3D referimos al apéndice A, al final del documento.

[9] Voxeles: Unidades mínimas de representación volumétrica de objetos.

4.2 – MAPAS DE TEXTURA Y DESPLEGADO DE COORDENADAS DE TEXTURA

Aunque ya tengamos la malla que define la superficie de nuestro objeto tridimensional, es probable que su aspecto no sea muy realista pues su superficie en sí es completamente lisa y uniformemente coloreada. La forma más sencilla de "pintar" una malla es asignar un color (generalmente en formato RGB[10], aunque hay otros como CMYK o HSV) a cada vértice. Para calcular el color de los puntos intermedios entre vértice y vértice se realiza una interpolación dependiendo de la distancia a los vértices de la cara a la que pertenece el punto de la superficie.

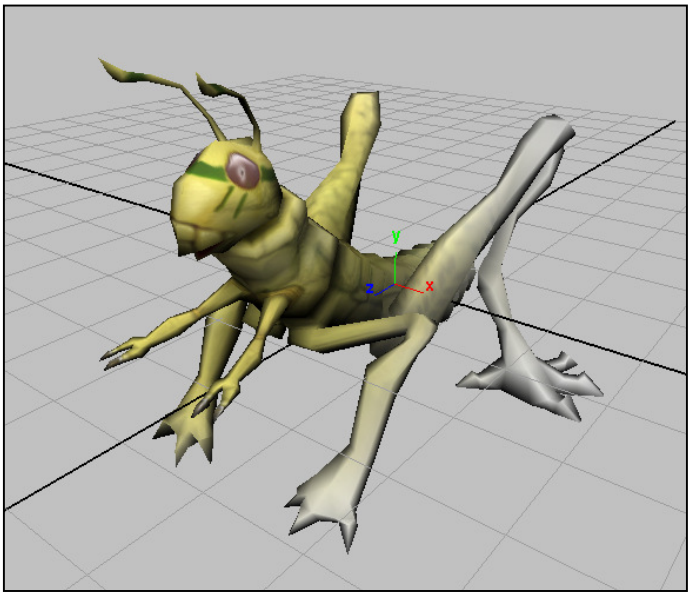


Imagen 8 : Una malla con textura de color difuso

Sin embargo, esta técnica, aunque simple, no es demasiado flexible ni ofrece un aspecto aceptable en la mayoría de casos. Para aumentar el detalle, se suele asignar punto a punto (comúnmente llamado "mapear") una imagen o un fragmento de ella a la superficie de cada polígono que representa el objeto tridimensional, de forma que el color de la superficie deja de estar limitado al valor interpolado de los colores de los vértices que componen el polígono y se toma de esta imagen o **textura**.

No solamente podemos variar de esta manera el color sino también la normal de la superficie y otras propiedades como el color reflejado (especular), transparencia, etcétera, pudiendo obtener de esta forma superficies con un aspecto rico y complejo.

[10] RGB: Formato de representación de colores mediante valores de intensidad de los colores rojo, verde y azul.

Imagen 9 : La textura plana del Saltamontes

En la imagen 9 podemos apreciar la textura del saltamontes que hemos utilizado para el ejemplo. Como se puede ver, podemos aprovechar para dibujar solo uno de los lados, ya que le podemos asignar los mismos fragmentos de imagen a los polígonos de ambos lados del insecto.

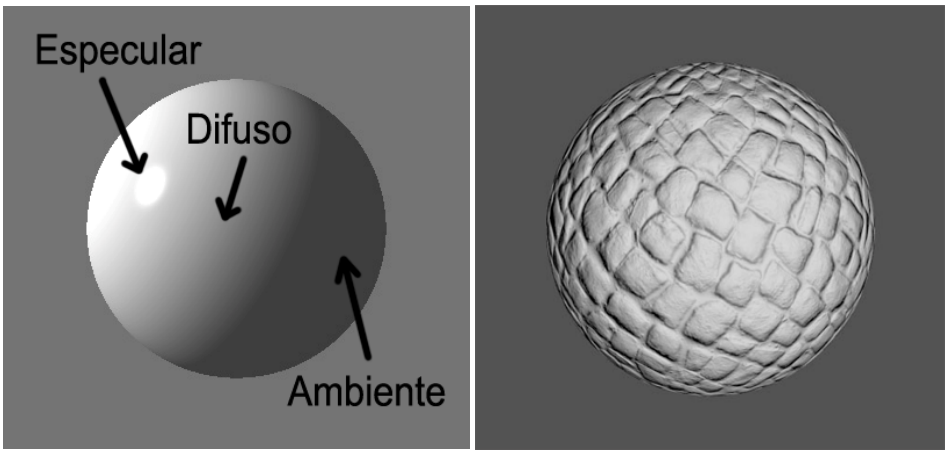


Imagen 10: Esfera sin mapa de normales y esfera con mapa de normales.

Para determinar qué fragmento de imagen corresponde a cada vértice de la malla se emplean las "coordenadas de textura", que asignan a cada vértice <x,y,z> una posición <s,t> (otra notación muy usada es <u,v>) en la imagen o textura. La asignación de estas coordenadas a cada vértice suele hacerse de manera manual o mediante algunos algoritmos de proyección que permiten acelerar el proceso. La realización de las imágenes que se mapearán sobre la malla suele hacerse a partir de fotografías, librerías y catálogos de texturas o de manera totalmente artesanal mediante un programa de edición fotográfica. En el caso de texturas que no representan color sino otras propiedades se emplean herramientas muy específicas destinadas a este fin (generadores de mapas de normales, "quemadores" de oclusión ambiental, etc).

4.3 - ESQUELETO, SKINNING Y SETUP

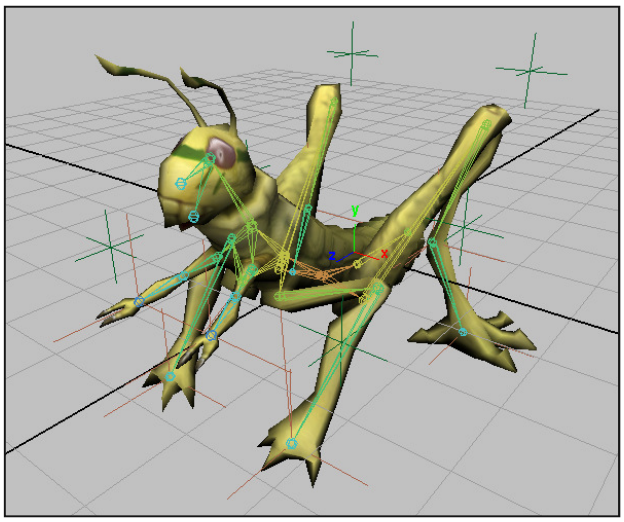


Imagen 11 : El modelo una vez incorporado el esqueleto

Aunque tengamos el objeto tridimensional ya creado, en muchas aplicaciones no es suficiente contar con una malla estática. Para dotar a la malla de animación, podemos usar dos tipos de métodos: los basados en interpolación de mallas y los basados en esqueletos.

4.3.1 - MÉTODOS BASADOS EN INTERPOLACIÓN DE MALLAS

Esta familia de métodos representa los diferentes cuadros de animación (o fotogramas clave) como mallas distintas separadas (varios personajes), que han de compartir el mismo número de vértices y que generalmente también comparten el desplegado de coordenadas de textura (uvs). La ilusión de movimiento se obtiene interpolando de manera secuencial cada malla con la siguiente. La ventaja que posee este método es que el artista posee control absoluto sobre el aspecto de cada cuadro de animación, no estando limitado de ninguna manera acerca de la forma o tamaño del objeto en cada cuadro, únicamente respetando la restricción de que todas las mallas han de poseer el mismo número de vértices para poder realizar la interpolación.

Una de las dos grandes desventajas que posee es que técnicamente requiere almacenar en la memoria del ordenador un gran volumen de datos acerca de cada malla lo que en la práctica hace que se utilicen sólo con mallas pequeñas o con un reducido número de cuadros de animación. La otra desventaja es que la animación no es fácil de alterar dinámicamente durante el juego, ya que a menudo requiere cálculos complejos sobre un gran número de vértices. Este método ha evolucionado en lo que se conoce como “**blend shapes**”, y se usa casi exclusivamente para animación facial.

4.3.2 - MÉTODOS BASADOS EN ESQUELETO

Los métodos basados en esqueleto son los más empleados hoy día por múltiples razones: Reducen muchísimo el empleo de memoria comparados con la interpolación de mallas, son mucho más flexibles y permiten efectos de animación más complejos, entre algunas de sus ventajas.

La idea se basa en definir para la malla que se quiere animar un conjunto de posiciones espaciales (que llamaremos juntas) unidas mediante "huesos" estableciendo una relación jerárquica entre ellas, de manera que las posiciones y rotaciones de cada junta están expresadas respecto al origen de coordenadas definido por la junta inmediatamente superior en la jerarquía. El conjunto de juntas y huesos se denomina el "**esqueleto**" de la malla, y suele seguir de forma más o menos precisa la morfología del objeto que se quiere animar. Una vez creada esta estructura, que podemos manipular como si de un armazón de alambre se tratara, se asigna a cada hueso un conjunto de vértices de la malla cuyo movimiento se verá influenciado por el del hueso. Para establecer la influencia que tenga el movimiento del hueso sobre el vértice, se usa un número decimal entre 1 (totalmente influenciado) y 0 (sin influencia) llamado **peso**.

A este proceso de asignar **pesos de animación** (que así se suelen denominar estos números) a cada vértice se le llama **skinning**, que viene de la palabra inglesa "piel", porque se asemeja a la piel de los seres vivos cuyo movimiento acompaña al de los huesos. Esta tarea se realiza principalmente, al igual que la asignación de coordenadas de textura, de forma manual con la ayuda de ciertos algoritmos de pesado automático.

Una vez se posee una malla asignada a un esqueleto que podemos manipular para deformar la malla como una marioneta, es común crear diversos manejadores y limitadores de movimiento al esqueleto para facilitar su uso en la posterior etapa de animación. A este nuevo proceso se le denomina **rigging o setup (preparación)**, y también es un trabajo eminentemente manual (Por ejemplo establecer límites de rotación para hacer que un antebrazo no pueda doblarse más de lo normal en un personaje humanoide).

En la imagen de ejemplo de la página anterior tenemos una de malla con su esqueleto, y con una serie de manejadores que en este caso se pueden visualizar en el programa como crucetas que facilitan la animación del personaje.

4.4 - ANIMACIÓN

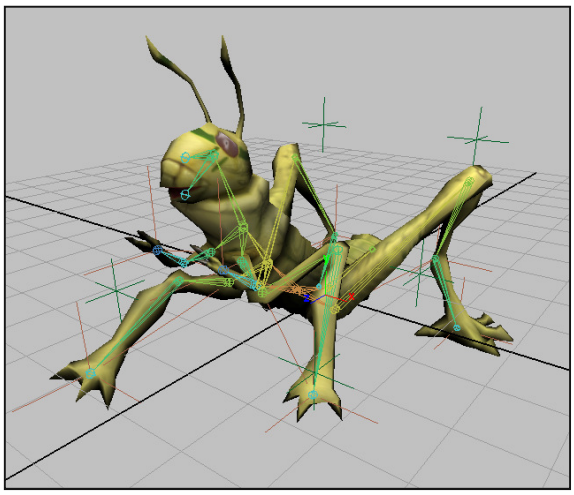


Imagen 12 : El modelo una vez animado (en uno de los fotogramas de la animación).

El esqueleto es colocado en una pose distinta para cada cuadro clave de la animación que se quiere conseguir y se almacenan las posiciones y rotaciones de cada junta para cada cuadro. Posteriormente a la hora de reproducir la animación se suele interpolar la posición de las juntas entre cada pareja de fotogramas definidos de manera análoga a como se hace en los método basados en interpolación de mallas.

La diferencia es que aquí se trabaja con un reducido número de posiciones y rotaciones para la animación (200 juntas para un esqueleto realmente complejo) en contraste con el elevado número de vértices para los que hay que almacenar datos al interpolar mallas (2000-5000 para un modelo modesto). Esto nos permite realizar operaciones más complejas sobre las juntas del esqueleto como corrección de rotación en tiempo real (se puede hacer que la cabeza y ojos de un personaje sigan a un objeto determinado, por ejemplo) o cinemática inversa, entre otras.

La colocación de las poses del esqueleto durante la animación se puede hacer de forma manual, tarea que lleva a cabo un equipo completo de animadores, o mediante técnicas de captura de movimiento, aunque no todo tipo de animaciones son susceptibles de ser capturadas del mundo real, evidentemente.

En resumidas cuentas, la creación de un avatar animado es siempre un proceso que requiere de mucho tiempo y esfuerzo, incluso para conseguir resultados mediocres. Las habilidades y conocimientos técnicos y artísticos y el equipo necesario para llevar a cabo esta tarea no están a disposición de cualquiera que pretenda embarcarse en la producción de una aplicación gráfica tridimensional. Esto es lo que trata de solucionar MAG.

5 – ESPECIFICACIÓN DETALLADA DE LAS HERRAMIENTAS

En este apartado detallaremos el diseño, la estructura y el formato de los distintos elementos que toman parte en el conjunto del proyecto. También explicaremos cómo utilizar el editor y los paquetes .mag.

5.1 - INTRODUCCIÓN

MAGTools (MAG) consigue acelerar el proceso de creación de avatares mediante la reutilización y almacenamiento de recursos de manera conveniente. Para tal finalidad, MAG ofrece un conjunto de herramientas, pero no es sólo eso, es un formato de intercambio de recursos basado en el formato COLLADA [11](COLLAborative Design Activity), ampliamente soportado por la industria.

El proyecto se compone de cuatro partes bien diferenciadas:

- Una librería dinámica (.dll) denominada Librería de Entrada (MAGInput).
- Una librería dinámica (.dll) denominada Librería de Salida (MAGOutput).
- Un formato de almacenamiento de recursos gráficos denominado paquete .MAG.
- Un editor 3D de paquetes .MAG.

Lo más importante que ofrece este sistema es el innovador flujo de trabajo, ya que permite diferir la generación de los avatares de la creación de los recursos gráficos empleados para su construcción. Hemos comentado anteriormente que una de las maneras que tienen los grafistas de acelerar el tedioso proceso de construcción de mallas, esqueletos, texturas y animaciones es reutilizar trabajo realizado anteriormente. Este trabajo no puede ser generado de manera procedural dado su carácter eminentemente artístico. Por lo tanto, no vamos a intentar automatizar su creación, si no acelerar su creación y evitar repetir trabajo que ya haya sido realizado anteriormente. MAG estandariza la creación de estos recursos mediante la Librería de Entrada, su almacenamiento mediante el paquete .MAG y su reutilización mediante la Librería de Salida.

El flujo de trabajo, en resumidas cuentas, es:

- Crear mallas, texturas, esqueletos y animaciones y exportarlos a formato Collada.
- Generar un paquete .MAG que contenga los recursos recién creados y añadir metainformación. (MAGInput).

[11] Collada: Formato de archivo para almacenar recursos gráficos. Ver Apéndice B.

- Emplear el paquete .MAG para generar personajes recombinando y transformando los recursos del paquete (mallas, esqueletos, texturas y animaciones), haciendo uso de la metainformación contenida en él. (MAGOutput).

Como hemos mencionado antes, el empleo de los paquetes de recursos es completamente independiente de cómo, quién o cuándo se crearon dichos paquetes, lo que permite a personas sin experiencia alguna en gráficos 3D generar personajes de manera sencilla y en poco tiempo.

A continuación detallamos una pequeña lista de características técnicas del proyecto:

- Soporte de mallas compuestas de polígonos de 3 y 4 vértices (triángulos y quads). Polígonos convexos de mayor número de caras no estás soportados, se recomienda triangular las mallas antes de su exportación a Collada.
- Soporte de blend shapes.
- Soporte de animación esqueletal multiinfluencia (hasta 4 influencias por vértice).
- Generación de personajes en formato .dae (Collada).
- Soporte de mapas de normales, oclusión ambiental y scattering subsuperficial.

Es necesario que el motor gráfico para el que estén destinados los personajes también soporte estas características o no podrá beneficiarse del soporte ofrecido por MAG en estas áreas durante el proceso de creación de personajes.

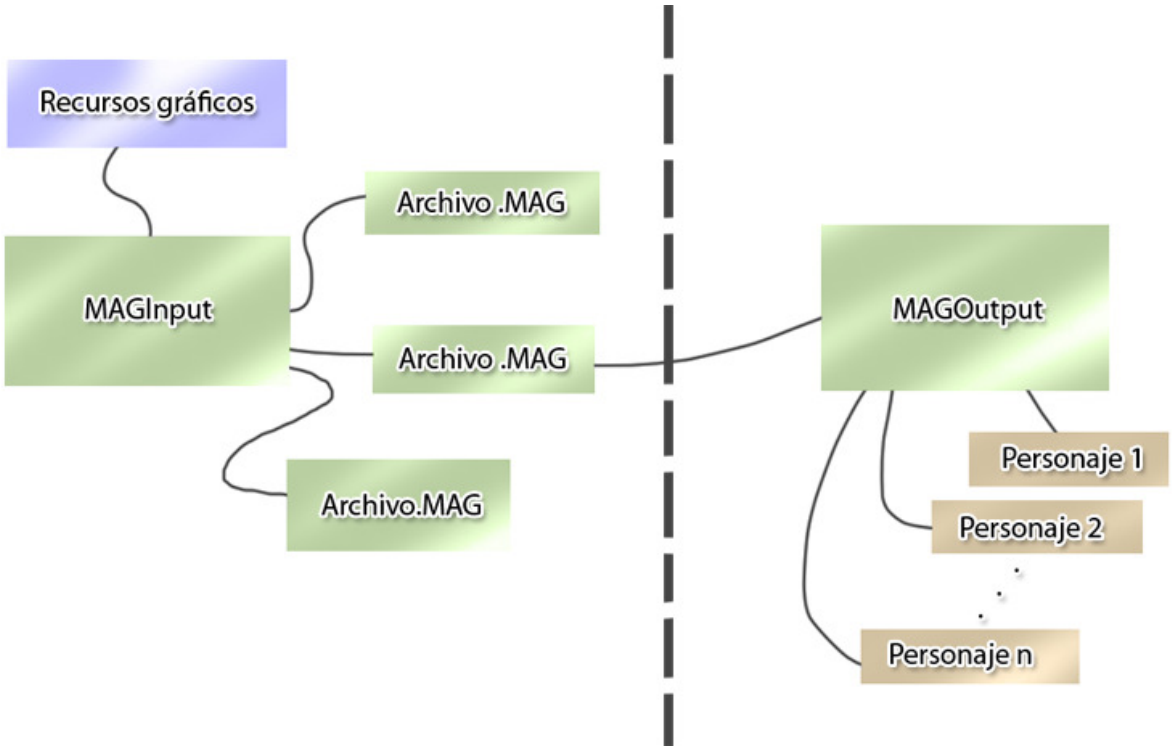


Imagen 13 : Elementos que toman parte en MagTools

5.2 - MAGINPUT : LA LIBRERÍA DE ENTRADA

La librería de entrada de MAG se encarga de, a partir de un conjunto de recursos gráficos y de información extra aportada por el usuario, generar un paquete que luego podrá ser utilizado por la librería de salida.

Para ello, cuenta con un API orientado al manejo y creación de paquetes, del cual vamos a esbozar su manejo a continuación: primero, se especifica que se va a crear un paquete y se convierte en el paquete "activo", sobre el que se realizarán el resto de acciones hasta que se cierre (se de por terminado) o se cree un paquete nuevo.

Una vez se tiene un paquete activo se puede comenzar a añadir recursos al paquete, organizados en entidades llamadas "personajes" ("characters"). Hay dos tipos de personajes: personajes "base" y personajes "variación". Ambos tipos de personajes pueden tener 4 tipos de recursos: Mallas, texturas, esqueletos y animaciones. Para añadir un personaje nuevo, se escoge un archivo de tipo collada (extensión .dae) y se especifica qué información vamos a extraer de él, entre los tres tipos posibles, o bien elegimos una textura.

El paquete puede contener tantos personajes "base" como se desee. Un personaje "base" puede, además de contener recursos, contener personajes "variación". No es posible incluir un recurso sin relacionarlo con un personaje, y no es posible incluir una "variación" sin relacionarla con una "base". Además, los personajes "variación" han de compartir una serie de características con el personaje "base" del que forman parte, a saber:

- Si poseen una malla, ha de tener el mismo número de vértices que la de la "base". Esto es esencial para que sea posible interpolar las mallas de base y variación, ya que si algún vértice de una de las mallas no posee un "análogo" en la otra malla, no poseemos información suficiente para interpolar su posición.
- Si poseen un esqueleto, ha de tener el mismo número de juntas (huesos) que el de la "base", por la misma razón por las que las mallas han de tener el mismo número de vértices.
- Si poseen una animación, ha de tener la misma duración que la de la "base".

Partiendo de un personaje "base", se puede mezclar de distintas formas con las "variaciones" que posee. De esta manera se establecen, de forma implícita, restricciones relacionales entre los recursos del paquete, ya que si el paquete está bien construido nunca será posible mezclar recursos técnicamente imposibles de interpolar, como por ejemplo dos mallas con distinto número de vértices.

5.2.1 – EL FORMATO MAG

Un paquete .MAG es un conjunto de archivos comprimidos en formato .zip. Estos archivos pueden ser de dos tipos:

- Archivo de recurso
- Archivo de metadatos (1 por paquete)

Los archivos de recurso, a su vez, pueden ser de dos tipos: .dae o archivo de imagen. Los .dae pueden contener información sobre una o varias mallas, esqueletos, y animaciones.

El archivo de metadatos, llamado "metadata.xml" es un archivo en formato XML que contiene información acerca de la manera en que deben ser usados los recursos gráficos contenidos en el paquete. Este archivo tiene la siguiente estructura:

- Un nodo raíz llamado <package>, que contiene como hijos una lista de < base_character >
- Nodos de tipo <base_character>, que representan un personaje base. Contienen nodos de tipo < variation_character> y varios tipos de recursos: <mesh>,<skeleton>,<animation>,<diffuse_map>,<normalspec_map>,<normal_map>,<scatteringocc_map>,<scattering_map>,<overlay1_map>,<overlay12_map>,<overlay123_map> y <overlay1234_map>
- Nodos de tipo <variation_character> que pueden tener los mismos nodos de recurso.

Un ejemplo de archivo de metadatos MAG:

```
<?xml version="1.0" ?>
<package>
    <base_character name="saltamontes">
        <variation_character name="cabeza_grande">
            <mesh path="salta_cab.dae" />
        </variation_character>
        <variation_character name="variacion_1">
            <mesh path="salta_var.dae" />
            <skeleton path="salta_varhueso2.dae" />
            <animation path="salta_varhueso2.dae" />
            <diffuse_map path="grasshopper_rebis0.tga" />
        </variation_character>
        <mesh path="salta.dae" />
        <skeleton path="salta.dae" />
        <animation path="salta.dae" />
        <diffuse_map path="grasshopper0.tga" />
    </base_character>
</package>
```

5.3 - MAGOUTPUT : LA LIBRERÍA DE SALIDA

Dado un paquete .mag con recursos gráficos, la librería de salida permite extraer los datos contenidos en el paquete a estructuras de datos manipulables por un motor gráfico, en este caso el del editor gráfico, y también el proceso inverso: tomar estas estructuras de datos, que pueden haber sido modificadas, y escribirlas a disco en formato Collada (.dae). Para ello posee varios métodos para obtener la estructura general del paquete, es decir, cuántos personajes base contiene y cuántas variaciones posee cada uno. Además, es posible recuperar información acerca de cada personaje (si posee o no un determinado tipo de recurso, como esqueleto o animaciones, por ejemplo) y extraer sus datos (malla, esqueleto, texturas, etc.) preparados para ser leídos por cualquier motor gráfico.

Veremos en la sección 7 el API en detalle de cada una de las librerías.

5.4 – MAGTOOLS: EL EDITOR

El editor que se ha desarrollado durante el proyecto tiene como finalidad principal la de servir de ejemplo de integración de las librerías utilizando nuestro propio motor gráfico y aplicando sus capacidades a todo el desarrollo de paquetes MAG y exportación y de herramienta gráfica para la creación y manipulación de paquetes .MAG, utilizando las funciones del API que describimos en puntos posteriores.

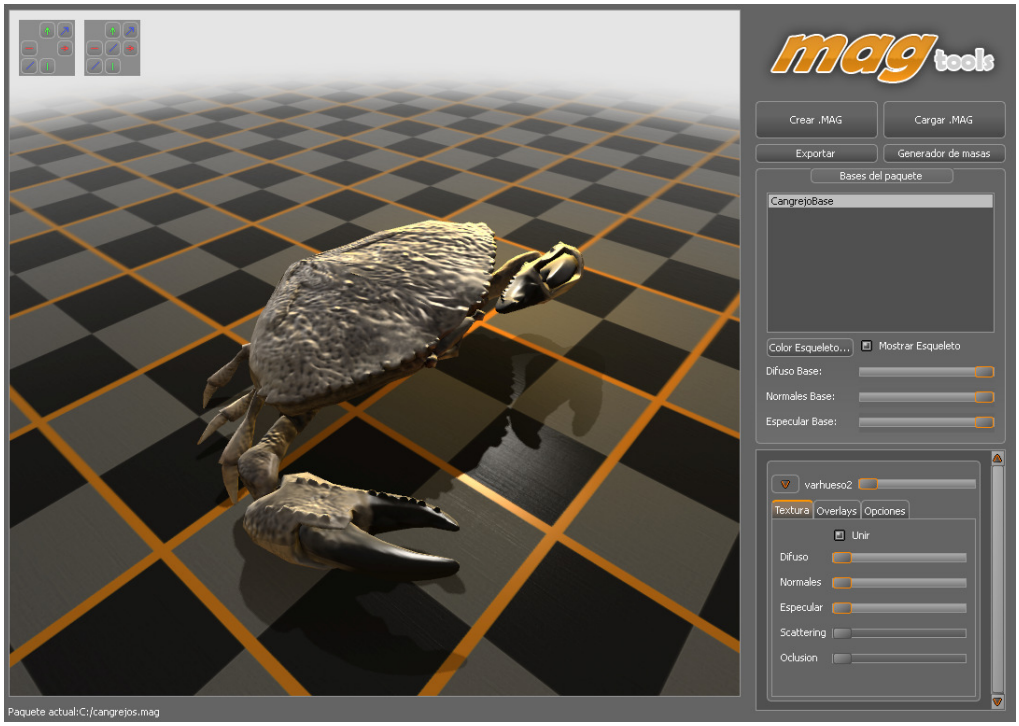


Imagen 14 : La pantalla principal del editor con un paquete (de cangrejos) abierto

Como puede verse en la imagen 14, consta de dos partes: a la izquierda, la **ventana de visualización** 3D y a la derecha el **menú de herramientas**.

Con él podremos realizar las siguientes tareas:

- Crear paquetes .MAG añadiendo personaje base y variaciones al personaje base, y añadirle recursos de todos los tipos desarrollados. Después exportar el paquete para crearlo y poder utilizarlo posteriormente.
- Abrir paquetes .MAG cargándolos en la vista para poder visualizarlo con detalle, y opcionalmente realizar cambios al personaje base.
- Exportar un personaje personalizado cada vez, o multitud de personajes siguiendo unos parámetros de mezcla utilizando distintos elementos del paquete. La modificación de un personaje en especial puede realizarse con las distintas pestañas que aparecerán bajo el listado del contenido del paquete, en la parte derecha.

A continuación incluimos una guía detallada de cómo utilizar los paquetes Mag en MAGTools.

5.4.1 – CÓMO UTILIZAR UN PAQUETE MAG EN EL EDITOR

Imagen 15 : Los botones principales de la aplicación

Al pulsar el botón “Cargar .MAG” el editor nos permite elegir un paquete y cargar sus contenidos para trabajar con ellos. Una vez escogido el paquete, en el área de “bases” del menú de herramientas, se puede seleccionar uno de los personajes “base” que contiene el paquete y empezar a trabajar con él.

Tras seleccionar el personaje “base”, éste aparece en la ventana 3D. La navegación por la escena 3D es muy sencilla:

- **Rotar la vista**: empleando el ratón, si se pincha y arrastra en cualquier lugar de la ventana 3D, se rota la cámara para mirar alrededor.
- **Desplazar cámara**: Mediante las teclas direccionales del teclado, se puede mover la cámara.

Además de aparecer el personaje en la vista 3D, también se ha rellenado en el menú de herramientas un listado de las variaciones disponibles para la base seleccionada en el cuadro de la derecha y en el espacio de abajo se nos muestran varios "widgets" cerrados que pueden ser desplegados utilizando la flecha abiertos para mostrar los controles de edición de esa variación.

Una vez abiertos, cada uno consta de un deslizador y tres pestañas: **Textura, Overlays y Opciones**. La pestaña "Textura" sirve para controlar la proporción de mezcla de los distintos mapas de textura, "Overlay" para controlar los 4 canales de color de la textura de tipo overlay, si la hay, y "Opciones" para cambiar parámetros de visualización. El deslizador principal (arriba del todo) controla el porcentaje de contribución de malla, esqueleto, animación y texturas de cada personaje "variación" a su personaje "base". Si está totalmente a la izquierda, se emplean los datos de la "base". Si está totalmente a la derecha, se emplean los de la "variación". Cualquier valor intermedio indica al editor que ha de calcular una interpolación entre ambos. Si se desactiva el checkbox "Unir", se puede controlar por separado el porcentaje de contribución de cada una de las texturas, siempre y cuando la variación las posea (en caso contrario su deslizador aparecerá en gris). Con "Unir" activado, los deslizadores de "Textura" vuelven a depender del principal.

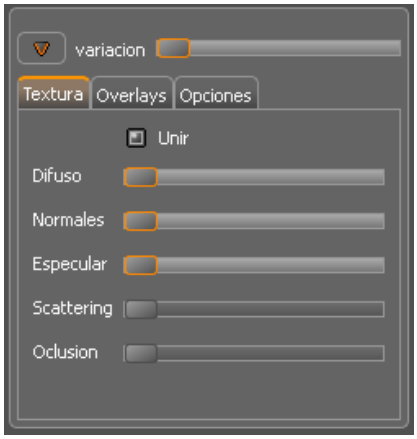


Imagen 16 : La pestaña textura de una variación

En la pestaña de Textura de la imagen 16 podemos indicar la cantidad de mezcla aplicada a cada uno de las 5 texturas que conforman el aspecto de la superficie del personaje:

- **Difuso**. Define el color real de la superficie del personaje
- **Normales**. Define la orientación de la superficie respecto a la luz (se pueden simular superficies rugosas y añadir detalle de esa manera).
- **Especular**. Afecta al nivel de brillo de la superficie.
- **Scattering**. Scattering cambia el color e intensidad de la luz que atraviesa el interior del personaje.
- **Oclusión**. Define las zonas del personaje donde llega menos luz como pliegues, zonas generalmente rodeadas de geometría…

Si alguna de las texturas no está disponible para esa variación aparecerá desactivada. Con el deslizador totalmente a la izquierda, no se aplica esa textura en absoluto, con él a la derecha, se aplica al 100%.

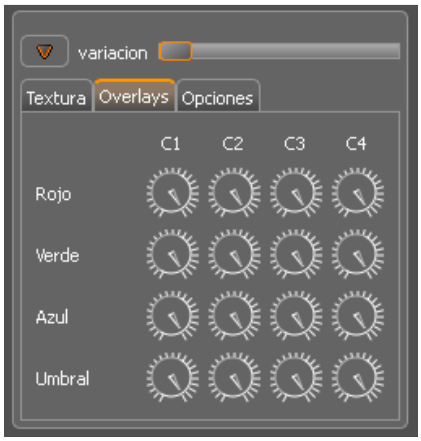


Imagen 17 : pestaña Overlays de una variación.

En la pestaña de Overlays de la imagen 17, tenemos controles para cambiar el aspecto del personaje mediante los 4 canales de “overlay” (o sobreimpresión). Son 4 imágenes en blanco y negro totalmente opacas que pueden ser teñidas de un color y poseen la capacidad de ser recortadas por sus bordes para cambiar su aspecto. Según cómo haya empleado el creador del paquete estas texturas, pueden servir para: alterar el motivo impreso en una tela, añadir tatuajes, heridas, suciedad, etc.

El color y el porcentaje de recorte de cada canal de overlay se controlan mediante la matriz de mandos.

Funcionan de la siguiente manera: cada columna de mandos representa uno de los 4 canales de overlay. Las tres primeras filas alteran la cantidad de color rojo, verde y azul del canal, respectivamente (mezclando estos tres colores es posible obtener cualquier color del espectro cromático, puesto que son los colores básicos). La última fila, “umbral”, determina a partir de qué intensidad (de 0 a 1) se muestra la imagen que contiene el canal. Inicialmente el umbral está al máximo (1), por lo que para mostrar el contenido del canal es necesario reducirlo a un umbral menor que 1.

En la imagen 18 puede verse un ejemplo, usando un mapa de overlay con color blanco en la camiseta (lo que será coloreado en este overlay), para el color de fondo completo. Después el segundo canal es el dibujo que aparece en rojo, a la izquierda, el tercero el dibujo que aparece en la espalda y en el cuarto coloreando de blanco las mangas y las líneas de la camiseta, con lo que se aplicaría el cambio de color en esas zonas únicamente. De esta manera una única textura rgba nos permite como mínimo cuatro diseños de camiseta (contando la camiseta lisa) cada uno con infinitas combinaciones de color. Dos texturas de overlay (una en cada variación) permitirían un mínimo de ocho diseños, tres texturas doce diseños y así sucesivamente. Si se activan al mismo tiempo varios canales, el número de combinaciones posibles crece exponencialmente.

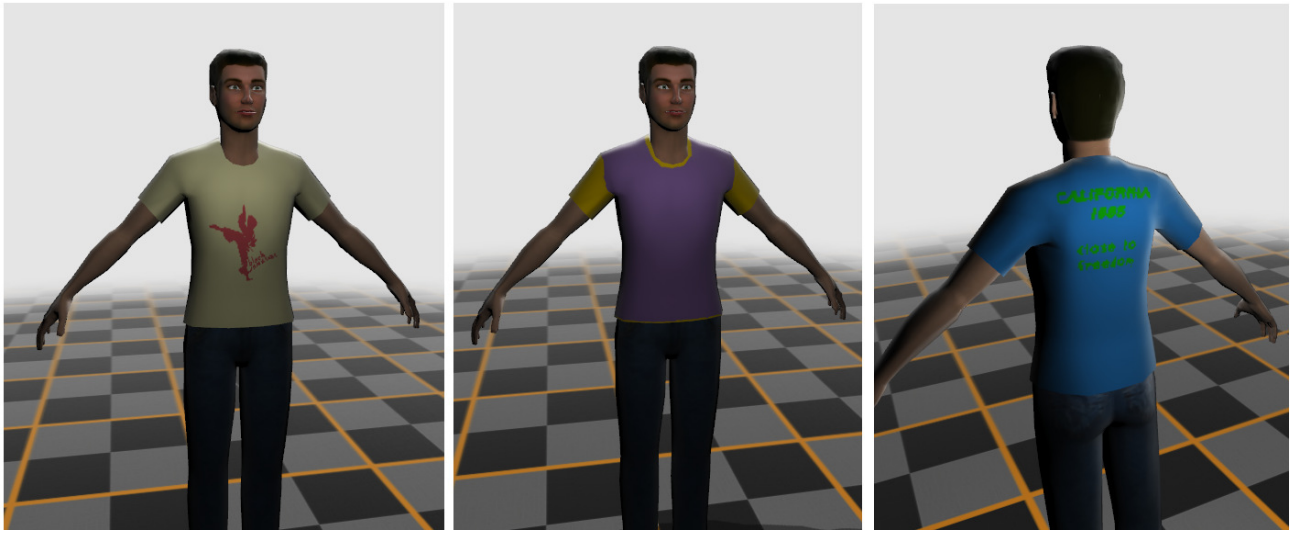


Imagen 18 : EL resultado de usar overlays de varias capas sobre la misma camiseta

En la pestaña Opciones de la variación, de la imagen 19, se puede hacer visible/invisible el esqueleto de la variación en la ventana 3D, y alterar su color (siempre y cuando la variación posea un esqueleto).

Cuando un personaje base tiene múltiples variaciones posibles, se pueden mezclar parámetros de varias de ellas. El editor se encarga de procesar los datos de mezcla mientras se modifica para verlo a tiempo real en la pantalla de visualización de la izquierda. Para que el resultado sea visualmente atractivo se realizan las siguientes tareas, información de utilidad para grafistas que se encarguen de esta tarea:

- Los porcentajes de mezcla de mapas de difuso y normales son normalizados para que la suma total entre todas las variaciones y la base sea 1: el porcentaje de cada textura es dividido entre la suma de todos. De esta manera se evitan artefactos de “quemado” (demasiado blanco) de la imagen, que ocurren cuando la suma de varias imágenes genera un resultado mayor que 1 en cuyo caso el resultado es indefinido aunque normalmente se produce el truncamiento a 1 de los píxeles afectados.
- Las mezclas de especular y de scattering no se normalizan, si no que se truncan entre 0 y 1.

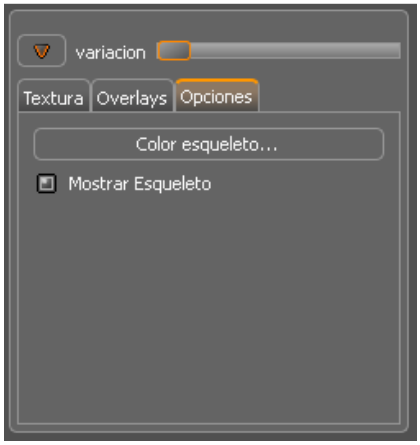


Imagen 19 : Menú de opciones de una variación

Finalmente, con los datos obtenidos de la mezcla de texturas de “base” y “variaciones”, se crean las cuatro texturas que exporta MAG por cada personaje: difuso/transparencia (difuso en rgb y transparencia en alpha), normales/especular (normales en rgb y especular en alpha), scattering/oclusión (scattering en rgb y oclusión en alpha), y overlays (1 canal en r, otro en b, otro en g, el último en alpha).

Por último, en la ventana de visualización disponemos del siguiente conjunto de desplazadores:

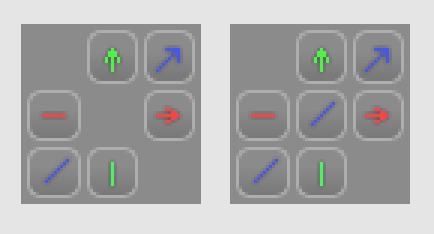


Imagen 20 : Botoneras para desplazar el avatar y la luz en el escenario principal

El cuadro de la izquierda tiene el objetivo de desplazar el avatar por la escena, por si su colocación original no nos permite verlo en su totalidad, o moverlo en otras direcciones. Las flechas rojas son movimiento en eje X, las flechas azules, eje Z, o profundidad, y las flechas verdes para desplazar en eje Y , o en altura.

El cuadro de la derecha sirve sin embargo para desplazar el punto de luz que está colocado sobre el avatar que ayuda ambientar el escenario y poder ver el resultado bajo la influencia de luces de algún color en especial. El botón central sirve para cambiar el color por otro cualquiera en un selector de color y el resto de botones tiene la misma finalidad que el cuadro de la izquierda: desplazar la luz a nuestro antojo.

6 – ALGORITMOS DE RENDERIZADO E INTERPOLACIÓN

Esta sección abarca todos los métodos y algoritmos que hemos utilizado para el desarrollo del proyecto. Se pretende explicar de forma razonada cada uno de ellos. Para hacerlo, se menciona en primer lugar el motor Hopp, desarrollado por José María Méndez con anterioridad, pues es usado para la aplicación de las técnicas.

En los siguientes apartados se usará lenguaje especialmente técnico perteneciente al campo de gráficos por ordenador. La intención es que sirva de referencia para cualquier desarrollador de aplicaciones tridimensionales que se encuentre en la misma problemática que nosotros, por lo tanto las explicaciones son muy pormenorizadas.

6.1- EL MOTOR HOPP

Para llevar a cabo el desarrollo de la parte gráfica del proyecto (visor 3D del editor) se ha optado por usar un motor gráfico propio dadas las ventajas, frente a un motor creado por terceros, de poseer el código fuente y de comprenderlo en mayor profundidad.

El motor se desarrolló con licencia LGPL[12] de manera independiente al proyecto de Sistemas Informáticos. Sin embargo, durante el desarrollo de MAG se han implementado mejoras. Principalmente dos:

- Animación esqueletal multi-influencia (cuatro influencias por vértice) con orientación de huesos.
- Blend shapes.

Además, ha sido necesario escribir un shader[13] de píxeles en GLSL[14] para poder representar en el editor la información contenida en los mapas de textura, de forma que puedan verse en tiempo real los cambios realizados a la superficie del personaje. En los siguientes apartados se explicarán en qué han consistido estas mejoras de manera detallada.

---

[12] Licencia Pública General Reducida, de GNU. Página web oficial: http://www.gnu.org/licenses/lgpl.html

[13] Conjunto de instrucciones programables por un desarrollador de una aplicación que son ejecutadas en la GPU, normalmente con el fin de realizar algún efecto al renderizar.

[14] OpenGL Shading Language. Más información: http://www.opengl.org/documentation/glsl/

6.1.1 – ANIMACIÓN ESQUELETAL

Al inicio del proyecto Hopp únicamente soportaba skinning[15] con una influencia por vértice. Ha sido necesario, para poder soportar el tipo de animación al que da soporta MAG y que se emplea en las aplicaciones de última generación (especialmente videojuegos), ampliar el límite de influencias a 4 y se ha trasladado el cálculo de la CPU a la GPU mediante un shader de vértices, por razones de eficiencia. De esta manera, cada vértice de la malla asociada a un esqueleto puede ser influenciado por hasta 4 huesos.

La animación del esqueleto se realiza en la CPU interpolando mediante varios métodos los keys (posiciones clave) de rotación de cada hueso, consiguiendo así un movimiento suave y sin saltos. Las rotaciones se almacenan internamente en el motor como **cuaterniones**[16] y sólo son convertidas a matrices 4x4 justo antes de aplicar la transformación a cada hueso.

Usando cuaterniones en lugar de matrices para representar las rotaciones se evitan problemas[17] como el bloqueo de Gimbal[18] (un cuaternión convierte una rotación Euler sobre 3 ejes x, y, z consecutivos en una sola rotación sobre un eje arbitrario, evitando por completo este problema), se reduce el espacio de almacenamiento (4 floats en lugar de 9) y se facilita la interpolación de rotaciones mediante el empleo de interpolación lineal esférica (slerp, de sus siglas en inglés, *spherical linear interpolation*).

Los vértices de la malla se envían sin transformar en espacio de objeto a la GPU, junto con atributos extra por cada vértice:

- Un vector tetradimensional que almacena los índices de los 4 huesos que afectan al vértice.
- Un vector tetradimensional que almacena el porcentaje (de 0 a 1) en que se ve afectado el vértice por cada hueso.
- Un array de matrices 4x4 que representa la información de transformación de cada hueso del esqueleto. Este array es accedido por el shader usando los índices almacenados en el primer vector tetradimensional.

De esta manera, una vez enviados los vértices sin procesar y la información de skinning, el cálculo efectivo del skin se realiza en la tarjeta gráfica. Este método mejora el rendimiento por dos razones:

- Se libera de trabajo a la CPU para trabajos que la requieran, ya que estos cálculos puede realizarlos la GPU a mayor velocidad.
- Se reduce el uso de ancho de banda de memoria al no tener que reenviar a la tarjeta gráfica un nuevo buffer de vértices en cada refresco de pantalla, si no que se envían una vez y es la GPU la que los transforma cuando es necesario.

---

[15] Skinning: Introducido y explicado en la sección 4.

[16] Cuaternión: Son una extensión de los números reales, así como los números complejos son una extensión de los reales por adición de la unidad imaginaria i, los cuaterniones son una extensión generada de manera análoga añadiendo las unidades imaginarias i, j y k a los números reales.

[17] Más información concerniente a este tema en Enlaces Útiles.

[18] Gimbal Lock: Pérdida de un eje de libertad. Referirse al enlace de la sección de Enlaces Útiles para más información.

Antes de realizar la animación de la malla es necesario almacenar las matrices de transformación de todos los huesos del esqueleto, estando éste posicionado en posición neutral, siguiendo la forma de la malla sin transformar. A este estado relajado se le denomina posición de **binding o base**. A las matrices de cada hueso en esta posición se las denomina **matrices de bind** o matrices base.

La fórmula para calcular la posición de los vértices de la malla durante el skinning empleando matrices de transformación en los huesos (hay una alternativa que es emplear directamente los cuaterniones, pero no permite interpolar la posición o escala de los huesos, únicamente su rotación) la detallamos a continuación. Dada una lista de influencias 0..i..n sobre un vértice v:

$$\mathbf{v}' = \sum w_i \mathbf{v} \cdot \mathbf{B}_{[i]}^{-1} \cdot \mathbf{W}_{[i]}$$

Ecuación 1 : Cálculo de influencias

En esta fórmula **v`** es la posición ya transformada del vértice. **Wi** es el peso de la influencia iésima sobre el vértice, **V** es la posición original del vértice en espacio de mundo. $B[i]^{-1}$ es la inversa de la matriz de bind del hueso iésimo en la lista de influencias y **W[i]** es la matriz de transformación acumulada en espacio de mundo de este hueso. De manera intuitiva, la fórmula expresa la nueva posición de un vértice como la suma de las transformaciones entre el estado base y el animado de cada hueso que afecta al vértice, pesadas con la influencia que tiene cada hueso sobre el vértice.

Este es el código (en GLSL, OpenGL Shading Language) del shader de vertices empleado para realizar en skinning en Hopp:

```
uniform mat4 skeleton[33];    //matrices de transformación de cada hueso.
attribute vec4 skeletonindex;    //para este vértice, índices en el array de huesos
attribute vec4 skeletonweights;    //para este vértice, pesos de cada hueso.

void main(void)
{
	gl_TexCoord[0] = gl_MultiTexCoord0;
	mat4 mat = 0.0;
	for (int i = 0; i < 4; i++) //cálculo del skinning
	{
		mat += skeleton[skeletonindex[i]] * skeletonweights[i];
	}

	gl_Position = gl_ModelViewProjectionMatrix * (mat * gl_Vertex);
}
```

El único detalle que merece ser mencionado es que en el array uniforme de matrices del esqueleto, cada matriz ya ha sido multiplicada por la inversa de la matriz de base, para evitar el paso de demasiados datos al shader. El resto es simplemente la implementación de la fórmula del skinning.

Otra de las características de la animación esqueletal en Hopp, que puede apreciarse en la imagen 21, es que ha sido necesario ampliar el almacenamiento de orientaciones iniciales para cada hueso del esqueleto. Muchos paquetes de animación no orientan por defecto los huesos con

respecto al origen de coordenadas del mundo, si no que los dotan con una orientación propia sobre la que se calculan las rotaciones y posiciones de la animación, una transformación base.

Esta orientación por hueso se añade como información extra al exportar los datos del esqueleto a un archivo, y si no es tenida en cuenta, al importar sus datos el resultado es completamente erróneo. Añadir información a los huesos de Hopp sobre su orientación inicial, sobre la que se aplica cualquier transformación posterior, fue suficiente para solventar este problema. La transformación rotacional final de cada hueso se calcula de la siguiente manera:

*MFinal = Mrotacion*MOrientacion*

A esta matriz resultante hay que añadirle el resto de transformaciones (escala, posición o rotación...).

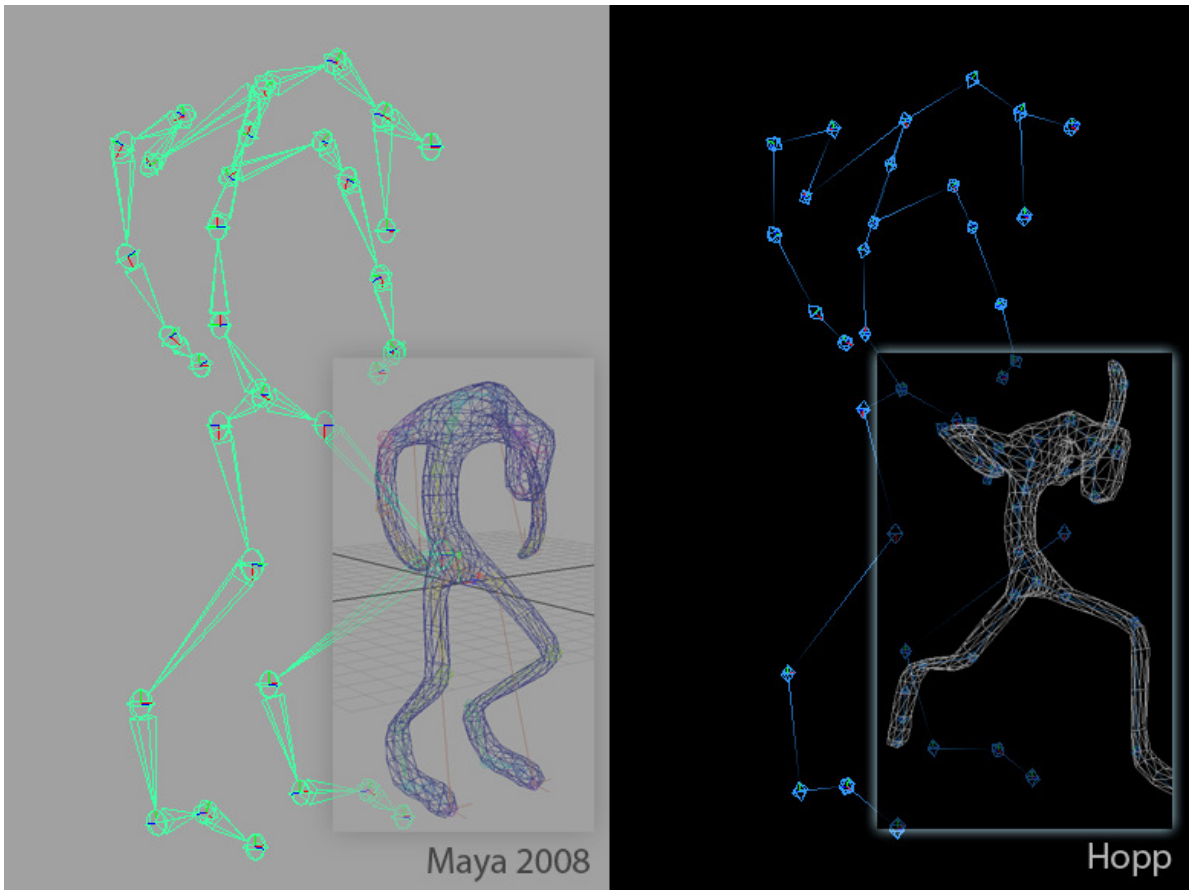


Imagen 21 : Diferencia de skinning del mismo personaje en Maya 2008 y en Hopp.

6.2- SHADING (SOMBREADO)

MAG almacena toda la información acerca del aspecto de la superficie del personaje mediante 4 mapas de textura:

- Mapa de Difuso/Alpha
- Mapa de Normales/Especular
- Mapa de Scattering/Oclusión
- Mapa de Overlays

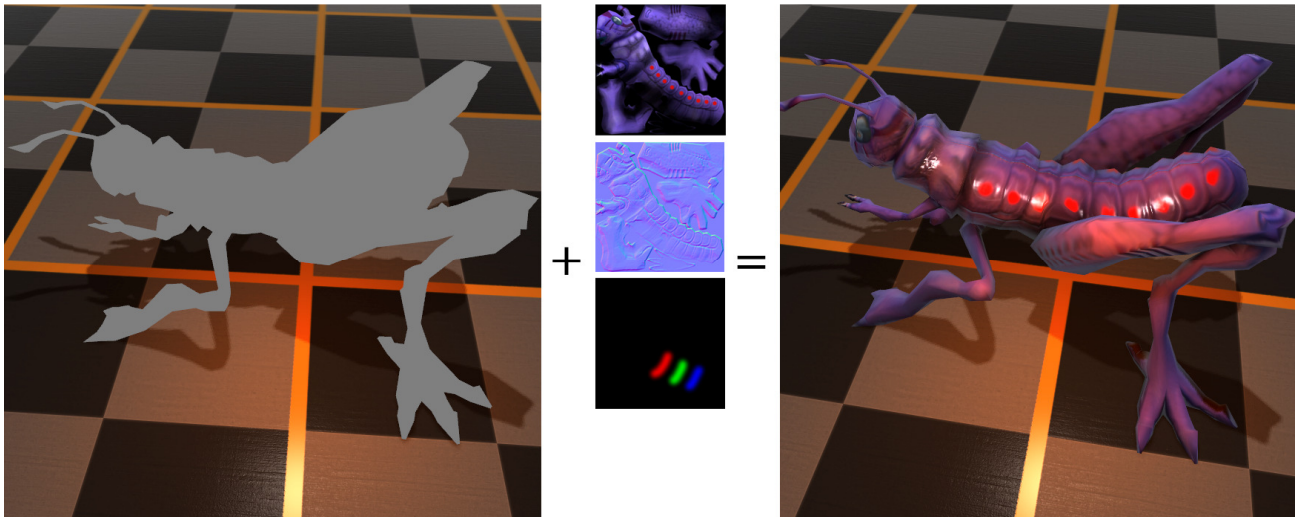


Imagen 22 : Texturas y modelo de iluminación.

Como introducción podemos ver en la imagen 22 como aplicándole al modelo sin texturizar a la izquierda varias texturas de diferentes tipos (de arriba abajo, textura de difuso, de normales y overlays) y un modelo de iluminación medianamente complejo es posible obtener resultados bastante atractivos visualmente.

Para poder mostrar en el editor -en tiempo real- los personajes con el aspecto apropiado, ha sido necesario escribir un shader de píxeles que lleve a cabo el sombreado e iluminación del personaje, basándose en la información recogida en estas 4 texturas, e información acerca de la posición, color e intensidad de la fuente de luz empleada. A continuación explicamos fragmento a fragmento el código GLSL, aunque no vamos a entrar en grandes detalles acerca de las técnicas empleadas ni de las fórmulas de los modelos de iluminación. Baste decir que como modelos principales hemos utilizado Phong [BUI1973] y Half Lambert [MIT2007], desarrollado por Valve.

Básicamente se calculan una serie de factores y con ellos se calcula el color final de cada píxel: color base, scattering o dispersión subsuperficial (difusión de la luz en el interior del personaje), factor de sombreado, iluminación directa, y factor de reflejo fresnel.Empezamos calculando la dirección del fragmento a sombrear hacia la luz (un vector unitario, ya que sólo nos indica dirección), y la distancia entre ambos, empleada para atenuar la luz:

```
vec3 tempLight= normalize(gl_LightSource[0].position.xyz - vertex);
float dist = length(gl_LightSource[0].position.xyz - vertex);
```

Acto seguido transformamos el vector dirección de la luz y el vector de cámara a espacio de tangentes, que es el espacio donde se harán los cálculos de iluminación siguiendo el modelo Phong. Este modelo requiere de tres vectores para calcular el resultado: normal de la superficie, dirección de la luz y dirección de la cámara.

```
mat3 tbn = mat3(normalize(vtangent), normalize(vbitangent), normalize(normal));

vec3 light = normalize(tempLight*tbn); //direction in tangent space
vec3 eye = normalize(-vertex*tbn); //camera vector in tangent space
```

Calculamos las regiones en sombra empleando shadowmaps (mapas de sombra) tradicionales con un filtro 3x3 PCF [REE1987] aplicado para suavizar los bordes de la región sombreada, dando lugar a una estrecha penumbra. Esta técnica de filtrado consiste en hacer la media de varias comparaciones binarias en una pequeña región del shadowmap, en lugar de quedarse con el resultado de una única comparación como sucede en el algoritmo original.

```
//shadows
float shadow = 1.0; //initialize as unshadowed
float depthmap = ProjSombra.z/ProjSombra.w;
vec2 proj = ProjSombra.st/ProjSombra.q;
shadow = 1.0-pcf(depthmap,1.0,proj);
float sd = distance(vec2(0.5),proj);
shadow += smoothstep(0.0,1.0,(sd-0.2)/0.3);
shadow = min(shadow,1.0);
```

La función que calcula el porcentaje de sombra mediante PCF (media de comparaciones) es la siguiente:

```
const float textureSize = 1024.0; //size of the texture
const float texelSize = 1.0 / textureSize; //size of one texel
float pcf(float z,float size, vec2 proj) {
float sum = 0;
int hf = PCF_NUM_SAMPLES/2;


for(int i=-hf; i <= hf; i++){
    for(int j=-hf; j <= hf; j++){
        sum += min(0.001,z-texture2D(ShadowMap,
        proj+vec2(i*size*texelSize,j*size*texelSize)).x)/0.001;
    }
}
return sum/(PCF_NUM_SAMPLES*PCF_NUM_SAMPLES);
}
```

Hay muchísimos artículos dedicados a intentar solventar los problemas y artefactos que sufren los shadowmaps. Uno de los métodos desarrollados más recientemente, que permite filtrar el shadowmap antes de realizar la comparación que determina el nivel de sombreado de cada píxel (lo cual ofrece importantes ventajas frente a otros métodos), son los VSMs [DON2006] basados en cálculos probabilísticos, que también consideramos emplear dada la sencillez de su implementación

pero de nuevo por falta de tiempo, han quedado fuera del proyecto siendo reemplazados por el PCF que acabamos de mostrar.

Tras calcular la cantidad de sombra sobre el píxel, recuperamos las normales y el canal de intensidad especular contenidos en las texturas de MAG:

```
vec3 n = normalize( texture2D(NormalMap, gl_TexCoord[0].st).xyz * 2.0 - 1.0);
float spec = texture2D(NormalMap, gl_TexCoord[0].st).a;
```

Es importante notar que las normales han de ser multiplicadas por 2.0 y después restar 1.0 para llevar cada componente del rango [0,1] en el que están expresadas en la textura correspondiente a [-1,1]. De no hacer esto, todas las normales estarían desplazadas y el resultado de la iluminación sería incorrecto.

Con todos los datos necesarios recuperados (distancia a la luz, dirección a la luz, vector normal, vector de cámara, intensidad especular) calculamos los coeficientes de iluminación, basada en el modelo de iluminación Phong:

```
//diffuse light
vec3 l_diffuse = clamp(dot(light,n),0.0,1.0)/(0.2*dist)*lightColor.xyz;
//specular light:
vec3 l_specular = pow(max(dot(reflect(-light,n), eye), 0.0), clamp(spec,0.1,1.0)*26.0);
//diffuse base color:
vec4 diffuse = texture2D(DiffuseMap, gl_TexCoord[0].st);
```

Calculamos el scattering subsuperficial -empleando un truco derivado del modelo de iluminación Lambert en lugar de los métodos físicamente correctos, mucho más intensivos computacionalmente- y el factor fresnel:

```
//fresnel rim:
float fresnel = clamp(dot(eye, n), 0.0, 1.0);
float shine = pow(1.0 - fresnel * fresnel , 5.0);

//scattering:
vec3 scatter = (clamp( max(0.0,dot(light,-n)) ,0.0,1.0)*halfLambert(-eye,light)/(0.5*dist)) *
vec3(0.8,0,0);
```

La función halfLambert se encarga de calcular el pseudo-scattering:

```
float halfLambert(in vec3 vect1, in vec3 vect2)
{
        float product = dot(vect1,vect2);
        return product * 0.5 + 0.5;
}
```

Por último, calculamos un coeficiente de luz difusa omnidireccional general, un coeficiente de luz ambiental hemisférica (completando un total de 3 luces: 2 omnidireccionales y una ambiental) y el

valor de color final, teniendo en cuenta todos los aspectos: sombras, iluminación phong, factor fresnel y scattering:

```
vec3 tempLight2= normalize(gl_LightSource[1].position.xyz - vertex);
vec3 dir_light = normalize(tempLight2*tbn);
vec3 dir_diffuse = vec3(clamp(dot(dir_light,n),0.0,1.0));
vec3 hem_ambient = mix(vec3(0.3,0.25,0.2)+0.2*lightColor.xyz,vec3(0.3,0.3,0.3),
(dot(dir_light,n)+1.0)*0.5);

vec3 final =
clamp(diffuse*(hem_ambient+dir_diffuse*shadow*0.3+scatter*lightColor.xyz+l_diffuse*0.5)+
l_specular*spec*shadow+shine*(dir_diffuse+l_diffuse)*0.6,0.0,1.0);

gl_FragColor = vec4(final,1.0);
```

Mediante el empleo de este shader, implementamos un modelo de sombreado que emplea toda la información proporcionada en el paquete MAG para conseguir un resultado realista, aunque por supuesto, una vez exportado el personaje a otros programas o motores, cómo emplear esta información queda como responsabilidad de dicho programa: puede emplear el mismo modelo de sombreado aquí expuesto para obtener los mismos resultados, o uno propio.

6.3 – ALGORITMOS DE MEZCLA

En este apartado explicaremos los algoritmos de los que se sirve el editor para mezclar los recursos y datos suministrados por un paquete .mag.

6.3.1 – BLEND SHAPES

Dado que Hopp no soportaba blend shapes (interpolaciones vértice a vértice entre múltiples mallas) ha sido necesario implementarlas. El cálculo de las blend shapes se ha realizado en CPU. La fórmula para obtener una malla final a partir de la mezcla de varias es la siguiente:

$$\mathbf{P}_{final} = \mathbf{P}_{neutral} + \sum_{i=0}^{N} \mathbf{w}_i \cdot \Delta\mathbf{P}_i.$$

Ecuación 2 : Cálculo de blend shapes

La posición final de cada vértice es su posición neutral (sin shapes aplicadas) más la suma de la posición de ese mismo vértice en cada blend shape aplicada multiplicado por un peso, para determinar cuánto influye. El Blend Shape se debe hacer con mallas con el mismo número de vértices, y se presupone que los vértices están numerados en el mismo orden.

El resultado al aplicar este método a dos mallas cualquiera con el mismo número de vértices es un término medio entre ambas, en función del valor del factor de interpolación (el peso) que puede estar entre 0 y 1.

6.3.2 - INTERPOLACIÓN LINEAL DE ESQUELETOS Y ANIMACIONES

Cuando dos o más variaciones que poseen un esqueleto animado son mezcladas, el resultado se calcula interpolando linealmente la posición de cada hueso entre las posiciones de los esqueletos objetivos. No sólo ha de interpolarse la posición, sino también la orientación y la rotación de cada hueso.

Este proceso se lleva a cabo de forma muy semejante a las blend shapes. La única diferencia es que es necesario interpolar rotaciones y orientaciones, ambas representadas mediante cuaterniones. Con los cuaterniones comúnmente se emplean dos tipos de interpolaciones: slerp (spherical linear interpolation) y nlerp[19] (normalized linear interpolation). A pesar de introducir pequeñas inexactitudes (velocidad angular no constante a lo largo de la trayectoria), hemos usado esta última por ser más sencilla y al mismo tiempo, adaptarse mejor a nuestras necesidades: es una operación conmutativa, y más rápida de calcular.

[19] Nlerp: http://number-none.com/product/Hacking%20Quaternions/index.html

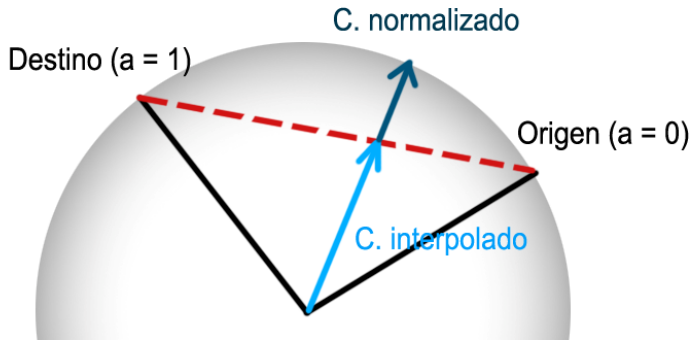


Imagen 23 : Interpolación y normalización

En la imagen 23, que muestra una esfera de radio unidad sobre la que se interpolan dos cuaterniones (líneas negras), vemos que la idea intuitiva es sencilla: interpolar linealmente los cuaterniones como si fueran vectores en R4, y normalizar el resultado para devolverlo a la superficie de la esfera. Por supuesto el resultado es incorrecto, ya que el desplazamiento real no se hace sobre la superficie de la esfera si no sobre una línea trazada por su interior entre el origen y el destino, lo cual causa las variaciones en la velocidad angular anteriormente mencionadas. Sin embargo la simplicidad de su implementación y su rapidez de cómputo hacen que sea preferible sobre slerp, ya que estas fluctuaciones de velocidad en la animación son prácticamente imperceptibles.

6.3.3 - PING-PONG DE BUFFERS Y ALPHA PREMULTIPLICADO

Cuando múltiples variaciones tienen influencia sobre el personaje final, sus mapas de textura han de ser mezclados entre sí.

Para poder llevar a cabo esta mezcla en el editor, en tiempo real, es necesario poder procesar paquetes de píxeles en paralelo, con lo que realizarla en la CPU no es posible. Las texturas han de ser cargadas en la memoria de vídeo de la tarjeta gráfica, y mezcladas empleando las unidades de procesamiento de flujo (stream processing units) de la tarjeta, es decir, empleando shaders. En nuestro caso no es posible emplear las funciones de mezcla que ofrece el pipeline por defecto de OpenGL (glBlendFunc) y que permiten realizar mezcla de transparencia directamente sobre un solo buffer, ya que difieren ligeramente de las que empleamos, y por lo tanto hemos de programarlas perdiendo la ventaja ofrecida por glBlendFunc.

Sin embargo, esto presenta algunos problemas: el primero, hay un límite al número de texturas que podemos procesar al mismo tiempo en la tarjeta gráfica, normalmente limitado por el número de unidades de textura disponibles, que puede estar entre 4 y 16 según el modelo de tarjeta... algunas veces más. Podemos solucionar esto tomando un buffer de memoria de vídeo y dibujando allí, de forma secuencial, una textura tras otra, mezclando cada una con el resultado de mezclar las anteriores. El problema de hacer esto reside en que no es posible leer y escribir simultáneamente en un buffer de memoria de la tarjeta gráfica, y por lo tanto no podemos llevar a cabo la mezcla de este modo.

El segundo problema es que la mezcla mediante transparencia (alpha blending) no es una operación asociativa. Esto quiere decir que, en general, dadas tres texturas A,B,C donde cada una posee 4 canales -rojo, verde, azul y alpha- y el operador de mezcla "+", (A+B)+C != A+(B+C). En nuestro caso significa que si nos limitamos a mezclar la textura n-ésima con el resultado de mezclar las n-1 anteriores, dependiendo del orden de mezcla obtendremos un resultado distinto. Esto obviamente no es lo que queremos. Existe una función de mezcla distinta al alpha tradicional, denominada alpha premultiplicado.

Hemos encontrado una solución a cada uno de estos dos problemas: Ping-Pong de buffers y alpha premultiplicado, respectivamente.

La técnica de Ping-Pong de buffers consiste en tomar dos buffers fuera de pantalla (regiones de memoria de vídeo donde podemos dibujar sin afectar a lo mostrado en pantalla) y alternar su uso como buffer de lectura y de escritura. Es un método idéntico al doble-buffering, con la salvedad de que en lugar de copiar datos de un buffer a otro, simplemente se intercambian los punteros a cada uno. Esto nos permite mezclar secuencialmente un número arbitrario de texturas, con una función de mezcla definida mediante un shader que toma el buffer de lectura como entrada, y presenta el resultado en el buffer de escritura. En pseudocódigo, el algoritmo presenta este aspecto:

```
buffer* bufferlectura;
buffer* bufferescritura;
para cada textura a mezclar:
        bufferescritura.escribir(textura);
        bufferescritura.mezclar(bufferlectura); //resultado se deja en bufferescritura
        buffer* aux = bufferescritura;
        bufferescritura = bufferlectura;
        bufferlectura = aux;
fin para
resultadofinal = bufferescritura;
```

Tratemos ahora la solución al segundo problema: históricamente, las bases para la mezcla mediante alpha premultiplicada fueron sentadas por Bruce Wallace y Marc Levoy en 1981 mientras trabajaban en los estudios de Hanna-Barbera y presentadas en [WAL1981] en forma de fórmula:

$$Aout = (1 - (1 - Afgd) * (1 - Abkg))$$

$$Cout = (Cfgd * Afgd + (1 - Afgd) * Cbkg * Abkg) / Aout$$

Donde Cfgb y Cbkg son los canales rgb de la textura "en primer plano" y "de fondo", respectivamente, y Afgd y Abkg los canales de alpha de esas mismas texturas. 4 años más tarde, en Lucasfims se llegó a la conclusión de que multiplicando los canales r,g y b de la textura por el canal de alpha, la fórmula de Wallace y Levoy quedaba bastante simplificada:

$$Aout = Afgd + (1 - Afgd) * Abkg$$

$$Cout = Cfgd' + (1 - Afgd) * Cbkg$$

Esta fórmula, además, puede implementarse parcialmente empleando las funciones de mezcla por defecto de OpenGL y evitar así usar ping-pong de buffers, ya que el color de "primer plano" (el que

vamos a escribir al buffer) está multiplicado por 1 y el de "fondo" (existente en el buffer), por 1-alpha del primer plano:

glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);

Haciendo la premultiplicación de los canales rgb mediante un shader, obtenemos la función de mezcla deseada, que nos garantiza que el orden de mezcla de las texturas no altera el resultado. Hay otras ventajas al uso de alpha premultiplicado, pero la que nos interesa aquí es ésta.

Por supuesto, este método solamente sirve si se desea mezclar mediante transparencia. En nuestro editor, lo hemos usado para mezclar los canales de difuso y de normales, este último, con la salvedad de que el resultado es tratado como un vector y normalizado antes de su uso. Pero el canal de especular y el de oclusión se han mezclado empleando suma y multiplicación lineales, respectivamente: operaciones que, por naturaleza, son asociativas, pero pueden ser implementadas mediante GLBlendFunc().

El ping-pong de buffers finalmente se ha empleado únicamente para mezclar los canales de overlay, ya que la función de mezcla empleada no puede ser reproducida mediante el pipeline por defecto de OpenGL, ni puede ser implementada mediante alpha premultiplicado.

6.3.4 - CANALES DE OVERLAY

Los 4 canales especiales (canales de "overlay" o sobreimpresión) permiten llevar a cabo alteraciones al aspecto del personaje que no podrían ser realizadas con ningún otro tipo de textura. Permiten imprimir diversas formas y colores sobre la superficie del personaje, permitiendo al usuario "pintar" sobre él, pero manteniendo el control en manos de los artistas que han diseñado el personaje.

Para ello, cada uno de los cuatro canales R,G,B y A de la textura de overlay, que son imágenes en escala de grises, es teñido de un color a elegir por el usuario, y recortado mediante un umbral que oculta las partes de la imagen con una luminosidad por debajo de él. La función de mezcla empleada es:

*Aoverlay = lerp(0.0,1.0, max(0.0,Ioverlay-Umbral)*10.0);*

Cfinal = Cbase(1.0- Aoverlay)+ Aoverlay*Coverlay;*

Donde Ioverlay es la intensidad del canal de overlay (el valor que se lee de la textura), Umbral es el umbral de corte por debajo del cual el overlay no afecta al color final, Cbase es el color base sobre el que se aplica el overlay (normalmente el color difuso del personaje) y Coverlay es el color del que se quiere teñir el canal de overlay.

En un principio barajamos la posibilidad de emplear la técnica detallada por Valve en **[GRE2007]**, basada en recortar mediante un umbral de alpha de 0.5 una imagen que codifica un campo de distancias al borde de un motivo o imagen binaria. De esta manera, únicamente se dibuja la parte de la imagen que está en el interior del borde, el cual es aproximado mediante segmentos lineales consiguiéndose una definición de imagen prácticamente perfecta (decimos prácticamente porque las esquinas muy afiladas no pueden ser correctamente reconstruidas mediante un único campo de distancias) a cualquier resolución, incluso con resoluciones del campo de distancias relativamente pequeñas.

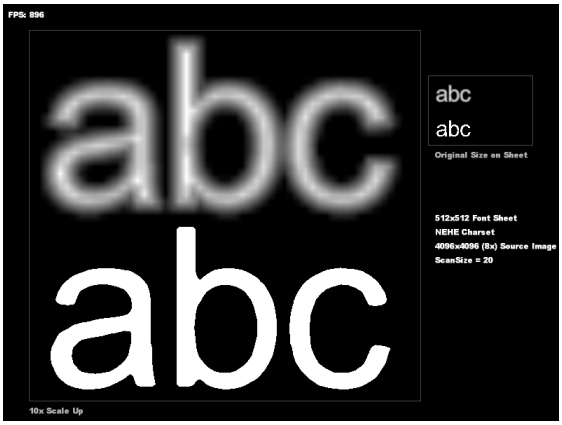


Imagen 24 : Campo de distancias y resultado.

En la imagen 24 podemos comparar el resultado con umbral de alpha a 0.5 abajo.

Sin embargo, debido a que esta técnica requiere un preprocesamiento de las texturas empleadas, y por lo tanto, la escritura de una herramienta de conversión de alpha a campo de distancias, la hemos dejado de lado por restricciones de tiempo. Aún así, si ya se dispone de una textura en este formato llevada a cabo con cualquier otra herramienta, gracias a la función de mezcla que empleamos, puede ser usada en el editor sin problemas.

7 – DOCUMENTACIÓN DEL PROYECTO

Hasta aquí hemos visto la teoría y algoritmos detrás de MAG, y su estructura externa en detalle. En esta sección se define el resultado del proyecto, así como el proceso de desarrollo. La primera parte de este capítulo será de especial interés para aquellos que quieran usar la herramienta desarrollada, pues se listan todas sus subrutinas.

7.1 – DISEÑO DEL PROYECTO

Cada uno de los tres componentes de MAG (las dos librerías y el editor) han sido pensados para ser eficientes, sencillos de manejar y extensibles en la medida de lo posible. Para ello se ha desarrollado al nivel más bajo posible, dando importancia a la facilidad de mantenimiento y de lectura del código.

7.1.1 - DISEÑO DE MAGINPUT

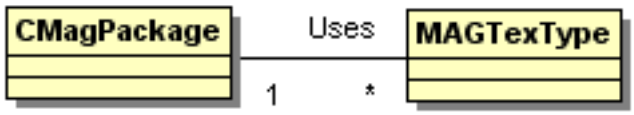


Imagen 25 : Diagrama de clases de MagInput

MAGInput es la más sencilla de las tres componentes del proyecto, y la primera que se comenzó a desarrollar. Consta de una única clase denominada CMagPackage. Esta clase representa un paquete .mag y su API pública es un puñado de funciones como siguen:

- void **addBaseCharacter**(const char* name)
- void **addVariationCharacter**(const char* basename,const char* name)
- void **addAnimationClip**(const char* charname,const char* path)
- void **addMesh**(const char* charname,const char* path)
- void **addSkeleton**(const char* charname,const char* path)
- void **addTextureMap**(const char* charname,const char* path, const MAGTexType& type)
- void **writePackage**(const char* path);

Todas las que comienzan por “add” (añadir) sirven para añadir un nuevo personaje o recurso al paquete MAG. Toman como parámetro el nombre del recurso y el path del archivo .dae del que se extrae.

Los personajes base (Base Character) son los elementos más altos en la jerarquía del paquete y en su interior contienen personajes variación, como ya hemos dicho anteriormente, además de

recursos. El resto de funciones tienen como objetivo añadir un recurso a un personaje determinado, identificado por su nombre.

La última función *writePackage* da por terminado el paquete, lo comprime y lo escribe a disco en el path que le indiquemos. Utilizará las siguientes funciones internas, además de otros recursos ocultos al usuario:

- zipFile **createZipFile**(const char* path);
- void **closeZipFile**(zipFile file);
- void **addFileToZip**(zipFile file,const char* name,const char* newname);

La única consideración de diseño que hemos tenido que tener en cuenta para esta librería ha sido el diseño del API, sencillo en extremo, de manera que cualquiera pueda emplearlo sin tener ningún tipo de conocimiento acerca del formato de almacenamiento de recursos ni de gráficos 3D. Aparte de esto, no hemos encontrado ninguna dificultad durante el desarrollo de esta librería, dada su simplicidad.

7.1.2 - DISEÑO DE MAGOUTPUT

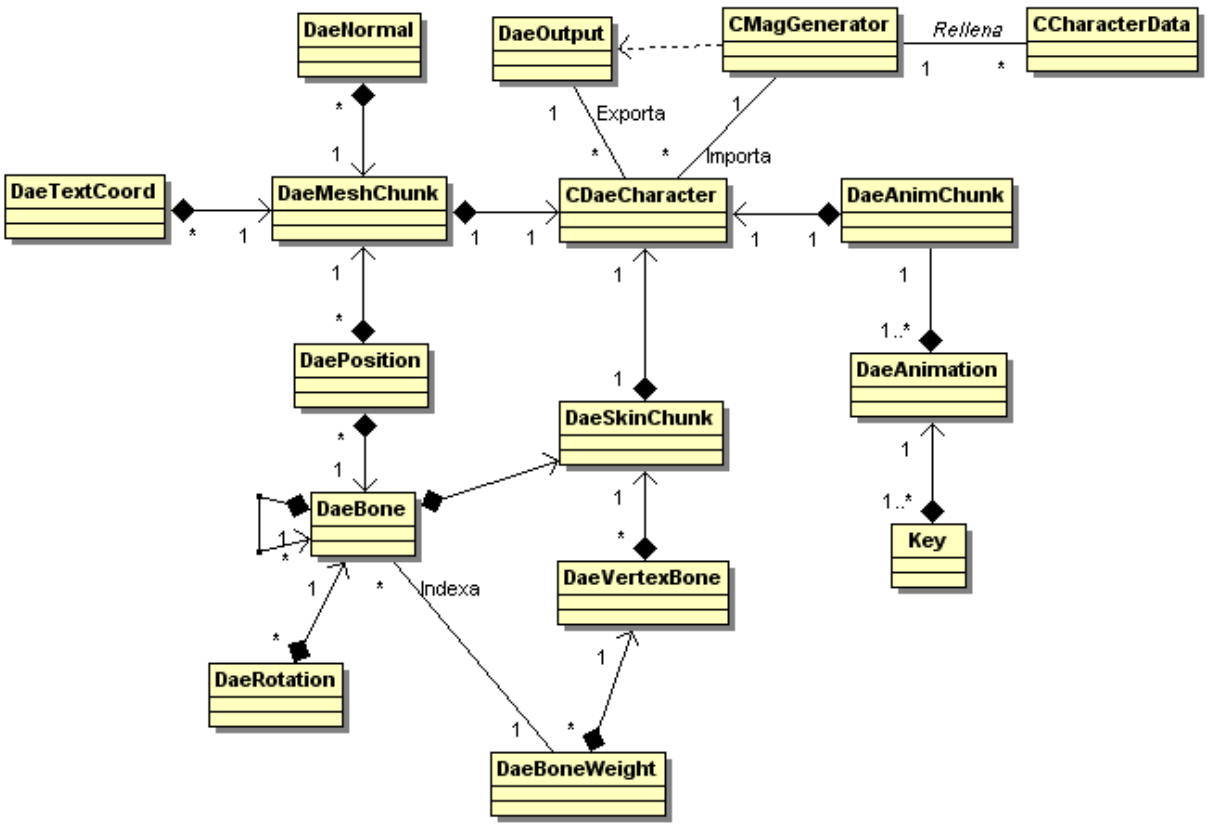


Imagen 26 : Diagrama de clases de MagOutput

MAGOutput es considerablemente más compleja que MAGInput, ya que es el núcleo del proyecto. Se trata de un conjunto de clases destinadas a leer archivos .mag, y para exportar archivos .dae creados a partir de la información leída del paquete. También contiene métodos para escribir a disco texturas extraídas directamente de la memoria de vídeo.

La principal dificultad encontrada durante el desarrollo del proyecto ha estado en esta librería y con el proceso de selección de la librería empleada para parsear archivos .dae del estándar Collada, de entre las distintas opciones existentes.

Existen principalmente dos librerías conocidas para leer los Collada: **FCollada** y **ColladaDOM**. A pesar de ser de más bajo nivel, nos decantamos por ColladaDOM ya que está mejor mantenida que FCollada y precisamente por permitir este control a bajo nivel, nos iba a brindar la flexibilidad necesaria.

A pesar de eso el soporte de ambas librerías y la documentación disponible es bastante limitada con lo que hubo dificultades para entender su API, trabajando la mayoría de las veces con las cabeceras de la librería y los comentarios añadidos por sus desarrolladores.

Ya que nuestra aplicación no necesita implementar todo el estándar Collada, únicamente se ha empleado un subconjunto, formado por la información básica relativa a mallas, esqueletos, y animaciones. Se transformaron a nuestros formatos internos de almacenamiento de recursos la información contenida en los archivos .dae obviando muchos datos inservibles para el propósito de las librerías, tales como la información de tipo de interpolación por cuadro de las animaciones (dado que nos hemos limitado a interpolar linealmente todas las transformaciones entre cuadros, la forma más sencilla). También se prescindió de la información referente a iluminación de la escena (se empleó una propia, como se ha descrito con anterioridad en la sección de Algoritmos), cámaras, superficies NURBS y datos exclusivos de la aplicación de la que procede el archivo .dae.

Hemos tomado como referencia a la hora de implementar la exportación e importación de este formato el generado por el plugin Collada Exporter de Maya 2008, y se ha comprobado la correctitud de los archivos que exporta nuestra librería empleando una herramienta conocida como Collada Coherency Test.

INTERFAZ MAGOUTPUT

En ella se definen los elementos principales y el API público de la librería. Además se definen las estructuras de datos intermedias de las que se habló en los párrafos anteriores.

El Api de Magoutput es el siguiente:

- **Estructura CDaeCharacter**:
  Establece los contenidos de un personaje. Contiene punteros a su malla, su esqueleto y su animación. Los tres son elementos de otras estructuras definidas (DaeMeshChunk, trozo de malla, daeSkinChunk , trozo de piel/esqueleto, daeAnimChunk, trozo de animación).
  Además contiene la información de difuso, normales, scattering y overlay, y sus longitudes.

- **Estructura CCharacterData**:
  Podemos con él construir una estructura que almacena una serie de booleanos y un nombre para la información de personaje. Los booleanos guardan si hay o no difuso, normales, scattering, especular, oclusión, overlays de alguno de los 4 tipos, malla y esqueleto.

- **Clase CmagGenerator**
  Implementa algoritmos para leer información de los .MAG, mezclando sus datos y finalmente exportando los personajes.

Su API pública es la siguiente, aparte de las constructoras y destructoras:

  - Bool **workWithPackage**(const char* path)
    Carga un paquete mag y lo establece como el paquete actual. Todas las demás funciones funcionarán con él. Si había uno abierto, se cerrará antes de abrir este otro.

  - std::map<std::string,std::vector<CCharacterData>> **getCharacterNamesTree**()
    Obtiene una estructura con la información de todos los personajes del paquete.

  - void **getCharacterFromPackage**(const char* name,CDaeCharacter* character)
    Obtiene un DaeCharacter con los datos de un personaje contenido en el paquete, a partir del nombre que lo identifica.

  - Bool **blendMeshes**(DaeMeshChunk* mesh1, DaeMeshChunk* mesh2,float amount)
    Sirve para mezclar dos mallas. Comprueba que ambas tienen los mismos polígonos y crea una nueva malla mezclando cada par de polígonos usando el valor de interpolación suministrado. La primera malla se rellena con el resultado de esta interpolación.

  - Bool **exportDaeCharacter**(const char* path, const CDaeCharacter* character)
    Genera un collada con la ruta dada y conteniendo el personaje aportado. En el caso de usar el editor a tiempo real y exportar el personaje creado, se iría modificando el DaeCharacter, y luego se utilizaría esta función para finalmente exportar las modificaciones que nos han gustado.

  Existen entre otras, algunas funciones privadas que son de utilidad para la lectura del paquete MAG, que ocultamos en el api:

  - bool **readFileFromPackage**(const char* name,int size_buf,char*buffer,int& readlength)

  - void **fillCharacterData**(TiXmlElement* elem,CDaeCharacter* character)
    (utiliza las tres siguientes funciones, que rellenan cada una de las partes del personaje)

  - void **readDaeMeshChunkData**(const char* path,DaeMeshChunk* chunk)
    (rellena el DaeMeshChunk con los datos obtenidos del .dae suministrado)

  - void **readDaeSkinChunkData**(const char* path,DaeSkinChunk* chunk)
    (rellena el DaeSkinChunk con los datos obtenidos del .dae suministrado)

- void **readDaeAnimChunkData**(const char* path,DaeAnimChunk* chunk)
  (rellena el DaeAnimChunk con los datos obtenidos del .dae suministrado)

INTERFAZ DAEREAD

Tiene todas las funcionalidades relacionadas con la lectura de los Dae y las nuevas y variadas estructuras de datos que luego utiliza el editor.

- **Estructura DaePosition**
  Es una estructura básica empleada para guardar normales o posiciones, todos aquellos datos que se componen de tres valores en punto flotante.
  Posee constructora propia y almacena los tres valores que se le proporcionen. Además tiene un método toString para obtener el texto y poder mostrar los datos a través de un stream.

- **Estructura DaeTranslation**
  Equivalente a DaePosition, es alias de la estructura anterior, empleado por claridad en el código. Se usa para guardar translaciones en el espacio (vectores, no puntos).

- **Estructura DaeRotation**
  DaeRotation almacena una rotación expresada como los cuatro valores reales de un cuaternión: a,b,c y d.

- **Enumerado TransformType**
  Es un enumerado que utilizamos para saber el tipo de transformación que llevamos a cabo, después de leerlo del Collada. Facilita la lectura del código. Contiene los tres tipos de transformación de rotación ROTATEX, ROTATEY, ROTATEZ, los tres tipos de transformación de translación TRANSLATEX, TRANSLATEY, TRANSLATEZ, y los tres de escala SCALEX, SCALEY, SCALEZ.

INFORMACIÓN RELACIONADA CON LAS MALLAS

- **Estructura DaeNormal**
  Equivalente a DaeTranslation pero para guardar posiciones de normales y identificarlo mejor en el código.

- **Estructura TextCoord**
  Guarda información sobre las coordenadas de textura, que sirven para colocar la textura, y están definidas de origen desde el programa 3d utilizado.
  Contiene dos float s y t, que son las coordenadas de posicionamiento, y una función para mostrar por pantalla los dos datos.

- **Estructura DaeMeshChunk**
  Implementa un contenedor para la información relacionada con las mallas utilizando las estructuras anteriores, y sacada del Collada .dae. Se almacenarán distintos “pedazos” o Chunks

según vayamos leyendo distintas estructuras definidas en el Collada, pues puede contener una malla definida de varias maneras en distintos nodos de tipo malla.

Dispone de contadores de distintos tipos de elementos que contiene, listas de posiciones, normales y coordenadas de textura, y arrays de índices relacionados con los anteriores para identificar sus posiciones.

Contiene además gran variedad de funciones públicas para trabajar con los contenidos dada la cantidad de información que guarda.

- Una lista de funciones de inicializar los datos de posiciones, normales, coordenadas de textura, índices de posiciones, índices de normales e índices de coordenadas de textura.
- Una lista de funciones para añadir posiciones, normales, coordenadas de textura, e índices de cada uno de ellos.
- Una lista de funciones que obtienen datos de los seis tipos de elementos indicados.
- Una lista para obtener el número de elementos de cada uno de los tres tipos, más funciones adicionales para saber algunos datos de conteo adicionales.

INFORMACIÓN RELACIONADA CON LOS HUESOS Y EL SKINNING

- **Estructura DaeBoneWeight**
  Estructura que contiene dos índices, para indicar un hueso y un peso.

- **Estructura DaeVertexBone**
  Simula lo que es la relación entre los vértices de la malla y el hueso dado un peso, para ver la influencia que tiene en él.
  Contiene una lista de DaeBoneWeight de relaciones de un vértice específico con que hueso, dado un peso.
  Sus funciones públicas solo permiten la consulta de las relaciones. Con los índices respectivos, se debe consultar a DaeSkinChunk para obtener los valores reales.

- **Estructura DaeBone**
  Esta estructura contiene la información necesaria de almacenar para cada hueso de un esqueleto.

  Contiene su nombre y el nombre de la junta relacionada al hueso (collada usa los dos para referenciar), un DaeTranslation con la translación que tiene el hueso, dos DaeRotation para almacenar la orientación y rotación del hueso, y una lista de índices de los huesos hijos del actual. Dado el índice del hijo, debemos consultar a DaeSkinChunk, que contiene todos los huesos.

  Y como funciones públicas contiene getters y setters de los datos indicados, además de funciones para obtener el índice de un hijo para poder movernos por el árbol jerárquico.

- **Estructura DaeSkinChunk**
  Esta es la estructura que contiene toda la información necesaria para hacer skinning a una geometría. Esencialmente, es un contenedor de toda la información necesaria. Principalmente contiene una array con todos los huesos definidos en Collada. Los huesos son de tipo DaeBone, y se pueden obtener tanto por índice como por nombre, aunque lo común es obtenerlo por índice. Además, se puede obtener los índices del hueso buscando por nombre y por nombre de junta. Para hacer estas operaciones (búsquedas por strings) se usa un hashmap de la STL[20].

  Además, collada da la opción a que exista más de un esqueleto que estén formados por subconjuntos de los huesos, indicando cual es el padre (recordar que los huesos están en una estructura de árbol). En esta estructura se almacena un array de índices a huesos que serían los nodos raíz de cada esqueleto. No es nada común que esto ocurra, por ello lo común es que este array solo disponga de un elemento.

  Como ya se ha comentado, almacena las relaciones que cada vértice tiene con los huesos. Para ello guarda un elemento de DaeVertexBone por cada vértice que tenga relaciones, que contendrá un array con todas sus relaciones a cada hueso. Hay un getter para devolver la estructura DaeVertexBone dado el índice del hueso.

  En caso que el índice del hueso sea -1, es porque está referenciando a la Bind-Shape Matrix[21], que también está contenida en esta estructura y se puede consultar como un array de 16 valores float. Los pesos están referenciados con índices a un array, donde se puede obtener su valor float real. Collada realiza esta simplificación para ahorrar memoria.

  Por último, se almacena la inversa de la matriz bind, necesaria para hacer los cálculos de las influencias, como se especifica en la sección 5.3.

INFORMACIÓN RELACIONADA CON LA ANIMACIÓN

- **Estructura Key**
  Almacena un key de animación, o key frame. Un key es una posición clave marcada en la linea de tiempo de la animación en cualquiera de los tipos de transformación de cualquier elemento del objeto animado. Marca un punto de referencia por el que los programas pueden interpolar entre distintos keys del mismo elemento, y realizar por si solo la animación.
  El archivo Collada almacena los key frames creados en el programa de 3d que utilicemos, e indica también el tipo de interpolación entre ellos.
  Nosotros precisamos por tanto guardar el tiempo en el que ocurre, el valor de la transformación en ese punto, y el tipo de interpolación, que nosotros hemos considerado solo que sea LINEAL, con vistas a un futuro próximo en el que ampliemos la variedad.
  Son la unidad básica en los DaeAnimationChunk. Se almacenan en listas.

---

[20] STL: Standard Template Library. Es una librería parcialmente incluida en el estándar de C++ que contiene una serie de contenedores y algoritmos útiles para acelerar las tareas de desarrollo.

[21] Matriz de Bind: Introducida en la sección 6.1.1.

- **Estructura DaeAnimation**
  Almacena un elemento básico de animación, un canal de animación, que es una colección de Keys de animación; un tipo de transformación (del enumerado definido al principio); y una juntura de hueso relacionada a la animación.
  En una animación completa tenemos muchos canales de animación como este, unidos completan la animación de todo el personaje.
  Además contiene una serie de funciones para obtener la información indicada, además de obtener un Key de la colección y cuántos Keys tiene, y poder añadir un key de animación, que es lo que hacemos al leer el Collada.

- **DaeAnimChunk**
  La estructura completa de la animación sacada de un nodo <animation> de Collada.
  Contiene únicamente una lista de DaeAnimation y podemos obtenerlas y añadirlas, además de imprimir la información por pantalla si es necesario.
  Sólo son públicas las funciones para obtener los datos, la escritura de datos no es necesaria hacerla porque esto lo realiza por si sola la librería al leer el Collada.

Como última explicación, existen tres funciones generales que son las más importantes para poder rellenar los objetos grandes indicados, (los Chunk), diciéndoles una ruta de archivo.

- **bool DLLE readDaeMesh(DaeMeshChunk* mesh,const char* filepath, const char* membuffer = 0)**
- **bool DLLE readDaeSkeleton(DaeSkinChunk* skin, const char* filepath, const char* membuffer = NULL)**
- **bool DLLE readDaeAnimation(DaeAnimChunk* anim,const char* filepath, const char* membuffer = NULL)**

INTERFAZ DAEWRITE

En este archivo disponemos de una estructura llamada DaeOutput.

Su intención es la de escribir un archivo DAE con la información que se le pasa. Se utiliza para exportar finalmente el personaje elegido, o la multitud de personajes repetidamente.

Transforma las estructuras Chunk definidas en el apartado anterior, en nodos del Collada para luego volcarlo en un archivo añadiendo los nodos restantes de escena y demás.

Disponemos de las siguientes funciones públicas del API:

- **void init(std::string path)**
  Es lo primero que debemos utilizar si queremos exportar un personaje. Inicializa el archivo Collada y guarda el nodo raíz para poder ir añadiendo a partir de ahora los datos.

- **bool endDaeOutput()**
  Esta funcion finaliza el archivo Collada añadiendo nodos adicionales que no tienen nada que ver con el personaje, pero sí con los huesos, que son en realidad puntos relacionados en la escena, y una escena donde colocamos los objetos. Es la última que debemos utilizar para finalizar.

- **void addDaeMesh(DaeMeshChunk* dmc,bool noskeleton)**
  Incluye el nodo de la malla en el archivo Collada obteniendo los datos del DaeMeshChunk. También necesita un indicador de si hay o no esqueleto para sus funciones internas y saber si debe generar cierta información.
- **void addDaeSkin(DaeSkinChunk* dmc)**
  Igual que la anterior función, incluye la información del skinning y las relaciones de pesos.

- **void addDaeAnim(DaeAnimChunk* dsc)**
  Igual que las anteriores, añadiendo el nodo relacionado de animación, y volcando toda la información en formato Collada.

INTERFAZ TEXTURECOMP

Es un archivo pequeño que contiene lo relacionado con la exportación de texturas utilizando la librería DevIL[22]. Es de uso interno.

Contiene dos funciones:

- **void initializeDevIL()**
  Como su nombre indica, inicializa la librería. Es necesario realizarlo antes de salvar la textura.

- **void saveOpenGLTexture(uint8_t* data,int w, int h,std::string filename)**
  Realiza la función de exportar la textura desde OpenGL. El tamaño de la imagen con los parámetros w y h , y un nombre de archivo final, obtenido del editor.

7.1.3 - DISEÑO DE MAGTOOLS

El editor MAGTools, que es una forma de mostrar un ejemplo de aplicación que utilizaría las librerías, es a lo que más esfuerzo en cuestión de aspecto estético y usabilidad le hemos aportado, para poder mostrar las enormes posibilidades que tiene nuestra librería unida a la capacidad gráfica de un motor de última generación. No tiene un tamaño mayor que MagOutput pero dado que incluye la librería Hopp sin compilar para poder modificar las fuentes a nuestro gusto según avanzamos lo que necesitemos, su tamaño final sí es mayor.

Poder mostrar las posibilidades de variación de personajes a tiempo real era nuestra intención desde el principio: generamos un paquete MAG utilizando unos menús que facilitan la tarea, que a su vez utilizan las funciones de MagInput. Después cargamos el paquete generado utilizando MagOutput y almacenando su información en estructuras de datos intermedias (definidas anteriormente) que permanecen activas durante el tiempo que el usuario decida.

Con esta información podemos tomar dos caminos: utilizar directamente la librería para generar automáticamente personajes, que es la verdadera función de las librerías, o lo que constituía el

[22] Ver apéndice C, dependencias del proyecto.

principal interés de cara a la presentación del proyecto, que es modificar a tiempo real los parámetros de los personajes utilizando los materiales definidos en el paquete MAG y utilizando las funciones de MagOutput con el intermediario de la interfaz gráfica de QT, además de los añadidos de aspecto estético que le dan un aspecto similar al de un videojuego o un simulador gráfico sin escribir un collada hasta que el usuario lo decida.

Por eso nos decidimos por diseñar una interfaz agradable, de sólido aspecto e intuitiva para que su aprendizaje fuese sencillo.

7.2 – PROCESO DE DESARROLLO DEL PROYECTO.

Para realizar el proyecto se ha seguido un esquema de desarrollo simple, compuesto por 5 iteraciones, intentando fijar unos objetivos parciales al final de cada una. Desde el principio del desarrollo había una idea clara de los objetivos que se querían conseguir y también de las tecnologías que se iban a emplear gracias a la experiencia previa desarrollando juegos y aplicaciones gráficas por parte de algunos miembros del equipo.

El esquema de las iteraciones que fue establecido al principio es el siguiente:

- Iteración 1:
    - Diseño de arquitectura/esqueleto.
    - Posibilidad de cargar mallas en MAGInput y MAGOutput, incluyendo la lectura desde Collada.
    - Realizar una interfaz de usuario experimental que emplee las librerías, como inicio del editor gráfico.
- Iteración 2:
    - Continuación del diseño general, en caso que fuera necesario cambiarse.
    - Carga de texturas básicas y coordenadas de textura tanto en MAGInput y MAGOutput.
    - Posibilidad de ver las texturas en el editor gráfico.
- Iteración 3:
    - Carga de esqueletos en MAGInput y MAGOutput.
    - Mejorar Hopp para permitir estas características.
- Iteración 4:
    - Carga de animaciones en MAGInput, MAGOutput y poder verlas en el editor.

- Iteración 5:
    - Mezclas de texturas y exportación de las mismas.
    - Overlays y añadidos gráficos.
    - Rediseñar el editor para mejorar su uso en caso que fuera necesario.
    - Realización de personajes de ejemplo.

Como puede verse, se ha trabajado en los 3 frentes del proyecto (librerías y editor) de manera horizontal, muchas veces solapando el fin de una iteración con el comienzo de la siguiente o incluso simultaneándolas durante un tiempo.

Al ser un grupo de desarrollo pequeño, esta flexibilidad nos ha permitido incrementar la velocidad de desarrollo sin perder el control sobre el proyecto.

A continuación se hace una descripción detallada de cada iteración:

Iteración 1

Puesta en marcha del proyecto. Esta iteración fue larga intencionadamente para permitir la investigación de las tecnologías y poder tomar decisiones sobre ellas.

- Fechas
    - Desde Octubre de 2009 hasta el 10 de Diciembre de 2009.
- Investigación:
    - Investigar el formato Collada, incluyendo las librerías para permitir su carga (explicadas anteriormente) y los exportadores en Maya de Collada.
    - Estudio y análisis de herramientas de compresión.
    - Aprendizaje sobre la API gráfica Qt.
- Desarrollo
    - Especificación inicial del formato MAG
    - Inicio de la implementación, preparación del proyecto.
    - Implementación de las librerías sólo para gestión de mallas.
    - Establecer interface para la generación de permutaciones.
    - Implementación básica del editor, sus ventanas iniciales.

La primera iteración terminó exitosamente y antes de lo esperado gracias a que ya conocíamos de antemano algunas de las tecnologías que íbamos a utilizar y teníamos una idea clara de la arquitectura.

Aquí nos decidimos por utilizar, como explicamos en la sección 6.1, después de probarlo y compararlo con otros, el parser ColladaDOM. Creamos una estructura inicial y diseñamos con QT una GUI para el editor que más tarde sería modificada.

En las librerías, entender Collada y ColladaDOM se pudo hacer sin mucha dificultad, aunque tuvimos algunas dificultades para poder depurar el código y conseguir que cargara todo correctamente.

En el editor, la facilidad de uso de QT hizo que avanzáramos deprisa, y pudimos integrar rápidamente el motor gráfico Hopp y las librerías, pudiendo generar un pequeño MAG con una malla y visualizarlo posteriormente con MagOutput a las pocas semanas de desarrollo.

Iteración 2

Esta iteración sirve para continuar con la implementación del proyecto. Con lo anterior bien hecho, debería ser bastante sencillo de implementar.

- Fechas
  - Desde el 10 de Diciembre de 2009 hasta el 19 de Enero de 2010.
- Investigación:
  - Investigación sobre la manera adecuada de gestionar las permutaciones y sus restricciones.
  - Investigación de la mejor manera de gestionar texturas y en qué formatos.
- Desarrollo
  - Implementación de permutaciones y sus restricciones.
  - Ampliar la funcionalidad de las librerías / editor para poder gestionar texturas.
  - Implementar la compresión de los paquetes.
  - Ampliar el editor para que sea capaz de cargar las texturas, e implementar controles básicos para permitir la edición.

La segunda iteración casi se hizo de forma seguida a la primera, excepto algunos detalles que los acabamos después de lo esperado.

La carga de texturas, como suponíamos, fue sencilla teniendo lo desarrollado en la primera iteración y sabiendo cómo funciona Collada al respecto. En el editor no hubo mucho problema para mostrar las texturas, y poder cargarlas no fue más que extender lo que teníamos hecho.

Iteración 3

La tercera iteración tiene bastante tiempo de desarrollo, pues es la iteración que asienta el proyecto. Su cometido no es solo continuar el desarrollo, si no dejar el proyecto estable para las siguientes iteraciones.

- Fechas
  - Desde el 19 de Enero de 2010 hasta el 9 de Marzo de 2010.
- Investigación:
  - Investigación necesaria para la gestión de esqueletos.
  - Establecer e implementar la estructura básica definitiva de los paquetes .MAG, tras sufrir múltiples cambios.

- Desarrollo

- Ampliar la funcionalidad de las librerías, editor y motor gráfico para poder gestionar esqueletos y leer el nuevo formato de paquete.
- Ampliar la funcionalidad del editor para que contenga todas las opciones que dispondrá la versión final, excepto las relacionadas con animaciones.
- Implementar todas las características necesarias en Hopp.

En la tercera iteración tuvimos más problemas de lo habitual. Para empezar, pensamos que en esas fechas estaríamos más libres de lo que realmente estuvimos. Esa falta de tiempo ocasionó que la implementación de algunos aspectos se retrasara. Al final conseguimos tenerlo todo para mediados de Marzo. Rediseñamos la estructura interna de los paquetes para simplificar tanto su creación como su uso, e introducimos los cambios necesarios en la arquitectura para soportar esta nueva especificación.

La carga de esqueleto y visualización de esqueletos también se complicó un poco más de lo esperado, debido a que Collada lo trataba de forma diferente a lo que esperábamos, y por ello llevo más esfuerzo.

Sobre el editor, se avanzó rápidamente pero muchas veces se rediseñó la interfaz para intentar hacerla más cómoda al usuario, lo que hizo que ocupara más tiempo del esperado.

Iteración 4

La cuarta iteración representa acabar con la parte más dura del proyecto, intentando dar por terminadas muchas características importantes dentro de nuestro proyecto.

- Fechas
  - Desde el 9 de Marzo de 2010 hasta el 15 de Abril de 2010.
- Investigación:
  - Investigación necesaria para la gestión de animaciones.
- Desarrollo
  - Ampliar la funcionalidad de las librerías / editor para poder gestionar animaciones.
  - Pulir el editor, principalmente en usabilidad, así como en rendimiento si se ve necesario.

Aunque la cuarta iteración, según lo planteado, es sencilla, fue una iteración bastante apurada. Esto fue debido a la cantidad de tareas pequeñas descolgadas de otras iteraciones que requerían ser atendidas. Se dedicó mucho tiempo a arreglar errores y de mejorar detalles. La mayoría de ellos eran fáciles de arreglar, pero como era una tarea que cambiaba de manos (uno pedía que arreglara esto a otro) tardábamos más de lo acostumbrado.

Iteración 5

La última iteración sirve para añadir las últimas características que no han sido implementadas en las anteriores iteraciones. Mucho trabajo de esta iteración es arreglar errores y corregir problemas pendientes.

- Fechas:
  - Desde el 15 de Abril de 2010 al 15 de Mayo de 2010.
- Desarrollo
  - Corrección de errores.
  - Mezclas de texturas y exportación de las mismas.
  - Overlays y añadidos gráficos.

La falta de tiempo ha sido el principal problema de esta iteración. Nos hemos retrasado en la implementación de algunas características, pero es debido a la gran carga lectiva que hemos tenido de muchas otras asignaturas. Más allá de eso, no hemos tenido ningún problema inesperado.

Después de la última fecha todavía nos sobra tiempo para terminar y depurar la aplicación, además de completar y terminar la memoria.

8 – CONCLUSIONES Y TRABAJO FUTURO

Aquí expondremos las conclusiones finales de la realización del proyecto, nuestra opinión sobre las herramientas y el desarrollo de la misma, y después un pequeño listado de posibles mejoras inmediatas interesantes para continuar el trabajo.

8.1 - CONCLUSIONES

Hemos presentado una aplicación que permite estructurar recursos gráficos tridimensionales y emplearlos para la generación de avatares preparados para su importación en cualquier motor o paquete de diseño compatible con el estándar Collada. La clave de esta aplicación es que cada avatar será diferente a los demás, aunque en el peor de los casos esta diferencia será sutil.

Nuestra aplicación hace uso de un conjunto limitado de recursos gráficos para generar un mayor reportorio de forma automatizada, ahorrando tiempo con ello. El resultado ha sido muy satisfactorio, pues hemos podido generar decenas de modelos diferentes totalmente funcionales.

Además, empleando las técnicas adecuadas, es factible realizar esto en tiempo real y mostrar una previsualización cercana al resultado final: iluminación, sombras, animación, etc. De esta forma nuestro proyecto se podría acercar al usuario final de forma totalmente intuitiva, permitiendo que el haga las modificaciones necesarias para personalizar su aspecto. La clave en comparación con otras herramientas es que no nos hemos centrado en una morfología concreta y el código desarrollado puede funcionar para cualquier tipo de personaje.

Creemos que el empleo de una herramienta de estas características, versátil y sin apenas restricciones en cuanto al tipo de avatares que puede generar, aceleraría el flujo de trabajo tradicional de creación de personajes tridimensionales.

A nivel profesional, el desarrollo de este proyecto nos ha permitido ampliar nuestros conocimientos de C y C++, de técnicas de almacenamiento y representación de datos en tres dimensiones, de álgebra matricial y modelos de iluminación en tiempo real, y de estructura y programación de unidades de procesamiento gráfico (GPUs).

Como experiencia personal, el grupo entero ha quedado satisfecho con el trabajo realizado, hemos aprendido a organizarnos y a establecer metras progresivas, con bastante éxito en opinión de los autores si se contempla en retrospectiva. Cada uno de los integrantes ha aprendido un poco de su compañero, y como caso destacado todos hemos aprendido de José María, debido a su profundo conocimiento sobre gráficos tridimensionales.

También queremos agradecer la ayuda prestada por los directores del proyecto, tanto de Pablo Gervás como de Gonzalo Méndez, por aceptar nuestro proyecto y confiar en nosotros para llevarlo a cabo.

8.2- TRABAJO FUTURO.

Una vez concluido el proyecto, hay múltiples aspectos que presentan margen para la mejora:

- La mezcla de mallas, esqueletos y texturas se realiza mediante hardware en el editor, y se recupera desde las librerías para su escritura a disco. A pesar de que emplear aceleración por hardware para llevar a cabo estas operaciones es un punto positivo (prácticamente indispensable), la librería MAGOutput en su estado actual carece de capacidad para mezclar recursos de forma autónoma, ya que depende de la aceleración suministrada por Hopp. La primera mejora que debería hacerse es integrar en la librería, implementados en software, los mismos algoritmos de mezcla empleados en el editor, para así hacer completamente independientes las librerías del editor y permitir su uso en sistemas carentes de o con pobre aceleración gráfica por hardware.

- Ya hemos mencionado anteriormente que se pensó en emplear campos de distancia para codificar los canales de overlay, llevar a cabo su implementación daría como resultado una mejora sustancial de la calidad de los overlays, disminuyendo la demanda de memoria.

- Se podría permitir el emparentamiento de mallas adicionales en determinados huesos de los esqueletos, o separar la ropa y complementos de las mallas base creando así un sistema de “capas” de geometría, todas animadas al mismo tiempo con el mismo esqueleto. De esta manera se reduciría el tamaño de los paquetes .MAG al eliminar geometría redundante y se facilitaría la creación de recursos gráficos de los paquetes. Esta mejora debería ser de las primeras en ser implementadas.

APÉNDICES

APÉNDICE A: ASPECTOS FUNDAMENTALES DE LOS GRÁFICOS 3D

Este apéndice es una breve introducción a algunos conceptos básicos necesarios para comprender los temas tratados en este documento. Supondremos por parte del lector conocimientos básicos de álgebra matricial y de vectores, ya que no trataremos la matemática necesaria con demasiada profundidad.

A.1 - REPRESENTACIÓN DE PUNTOS Y VECTORES

En un espacio tridimensional como el que nos ocupa (R3), podemos tener dos elementos básicos: un **punto (P)**, es decir, una posición dada por tres coordenadas X, Y, Z y un **vector**, que puede ser interpretado como un punto y una dirección o como la diferencia entre dos puntos. Tanto puntos como vectores se suelen representar mediante un vector matemático de 3 coordenadas, que en el caso del punto determinan su posición y en el caso del vector, su dirección, suponiendo que su origen es <0, 0, 0>. En la práctica la dirección y sentido de las coordenadas X, Y, Z depende del motor gráfico y/o API gráfica (OpenGL, DirectX[23]) que se emplee.

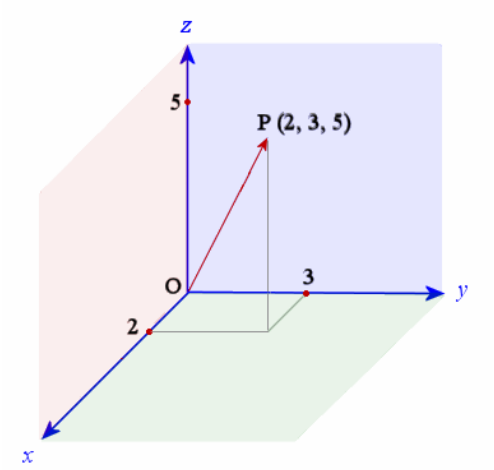


Imagen 27 : Representación gráfica de un punto en el espacio.

[23] OpenGL, DirectX: Principales APIs para desarrollo de aplicaciones con gráficos tridimensionales. OpenGL es mantenida por Khronos Group, mientras que DirectX es propiedad de Microsoft.

A.2 - TRANSFORMACIONES AFINES

Ambos elementos pueden sufrir **transformaciones** de sus propiedades en el espacio, principalmente tres:

- **Traslación:** Cambio de posición.
- **Rotación:** Rotación de un ángulo según un eje.
- **Escalado:** Modificación del tamaño dado un eje.

Estas transformaciones, denominadas transformaciones afines, se representan mediante matrices de 4 filas y 4 columnas que generan, al ser multiplicadas por un vector de 4 valores, un nuevo vector que es el original con las transformaciones correspondientes aplicadas.

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\operatorname{sen}\theta & 0 \\ 0 & \operatorname{sen}\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_y = \begin{bmatrix} \cos\theta & 0 & \operatorname{sen}\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\operatorname{sen}\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R_z = \begin{bmatrix} \cos\theta & -\operatorname{sen}\theta & 0 & 0 \\ \operatorname{sen}\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ecuación 3 : Matrices de transformación. De izq. a derecha, rotación en el eje X, Y, Z

$$T_{\mathbf{v}} = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Ecuación 4 : Matriz de traslación

¿Por qué un vector de 4 dimensiones en lugar de 3? La 4ª coordenada de los vectores se suele denominar coordenada homogénea (w), y es empleada para poder realizar transformaciones no lineales como la traslación, como si se tratase de una transformación lineal en un espacio de dimensión superior (4 en lugar de 3). Generalmente esta coordenada vale 1 y para nuestros propósitos podemos olvidarnos de ella. Consideraremos las matrices como “cajas negras” que aplican transformaciones a vectores, sin entrar en detalles de la construcción de dichas matrices.

Veamos en la imagen 28 un ejemplo de rotación, traslación y escalado en 2 dimensiones para facilitar su visualización (en blanco el objeto sin transformar, en azul ya transformado, y en rojo el desplazamiento experimentado por sus vértices)

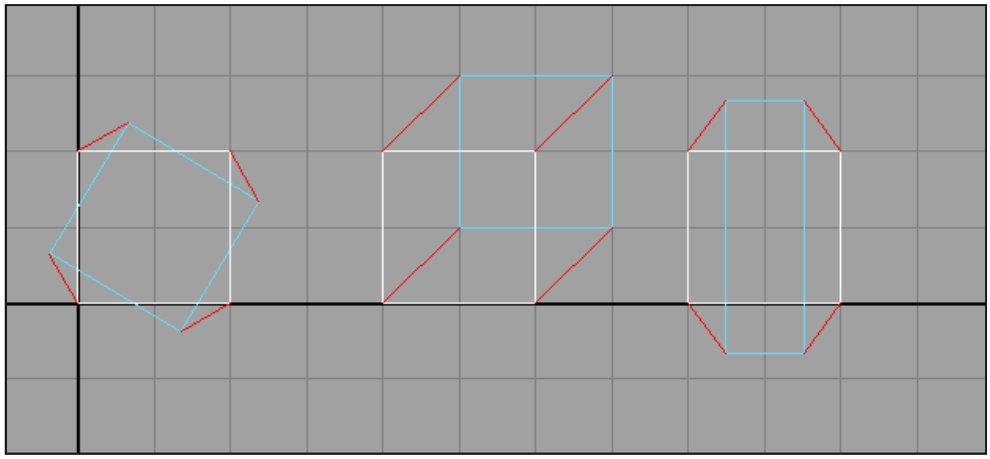


Imagen 28 : tres tipos de transformaciones afines.

Es habitual (y aquí se hará a menudo) **componer transformaciones**: rotar y trasladar, escalar y rotar, escalar y trasladar... y la manera de obtener estas transformaciones, resultado de componer las tres básicas, es multiplicar sus matrices de transformación. Así, la matriz que representa la transformación resultante de aplicar primero A y luego B es B*A. El orden de multiplicación es importante que como sabemos la multiplicación de matrices no es conmutativa (A*B no da el mismo resultado que B*A). Se puede seguir multiplicando el resultado por la izquierda todas las veces que se necesite con lo que se irán concatenando transformaciones una tras otra. Todas las transformaciones se hacen sobre <0,0,0> (se rota sobre el origen, se escala a partir del origen, etc).

En la imagen 29 ilustramos la importancia del orden de transformación. Si primero rotamos y después trasladamos, el resultado no es el mismo que se obtiene si trasladamos antes y rotamos después.

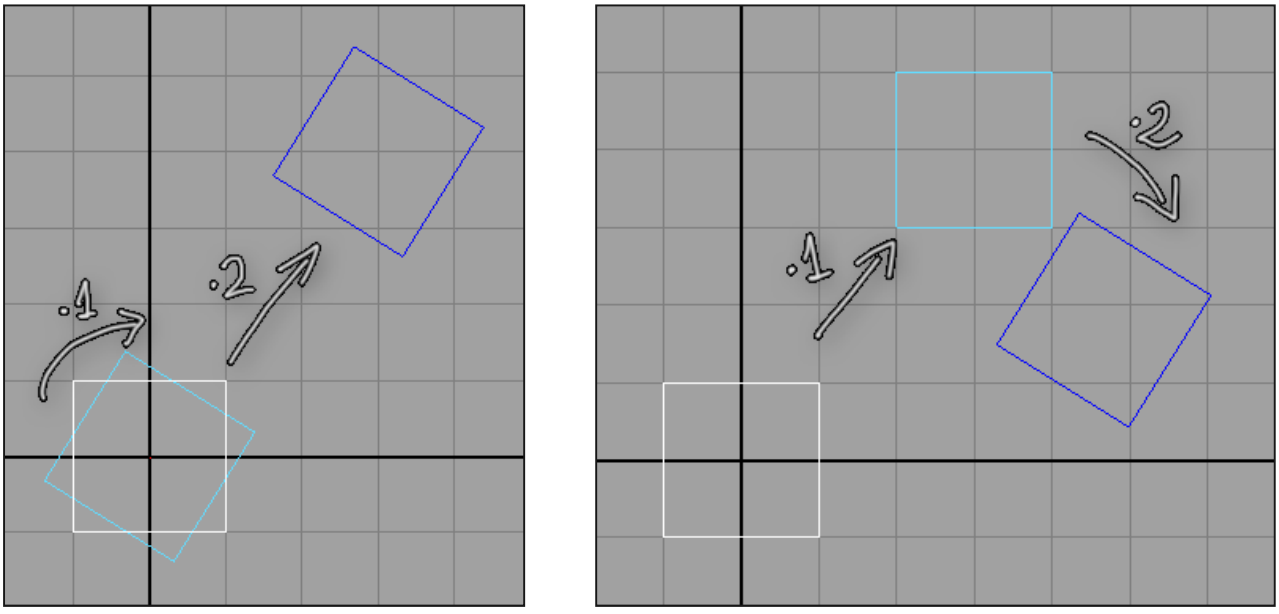


Imagen 29 : Componer transformaciones en orden diferente resulta en una figura diferente

A.3 - ESPACIOS

El siguiente concepto clave para entender este documento es el concepto de **espacio**. Un espacio, matemáticamente hablando, es el "marco" generado por un origen de coordenadas.

Todo punto o vector está expresado en función de un origen de coordenadas. Estamos acostumbrados a que éste esté centrado en <0, 0, 0> y que sus ejes sean de longitud 1, sin rotación alguna.

Pero esto no tiene porqué ser así. En un espacio tridimensional donde hay definida una jerarquía de objetos, se suelen distinguir 3 espacios: **Mundo, Local y Objeto**:

- El espacio de **mundo** es el definido por el origen de coordenadas centrado en <0,0,0>, al que todos estamos acostumbrados.
- Cada objeto define su propio espacio de coordenadas, en el que él está situado en el origen. A estos espacios propios de cada objeto se les denomina espacios de **objeto**.
- Si un objeto está definido en el espacio de otro (sus transformaciones están concatenadas a las del primer objeto), diremos que está **emparentado** al primero. En ese caso, el espacio definido por el objeto "padre", es decir su espacio de objeto, es el espacio local de su hijo.
- Existe un último espacio de uso común denominado "espacio de tangente", empleado principalmente en la iluminación de superficies.

Es posible obtener las coordenadas de un punto o un vector expresadas en otro espacio diferente a aquel en el que están expresadas originalmente, multiplicándolos por la matriz base o generadora del espacio de destino. A esta operación se le denomina **cambio de base** o **cambio de espacio**.

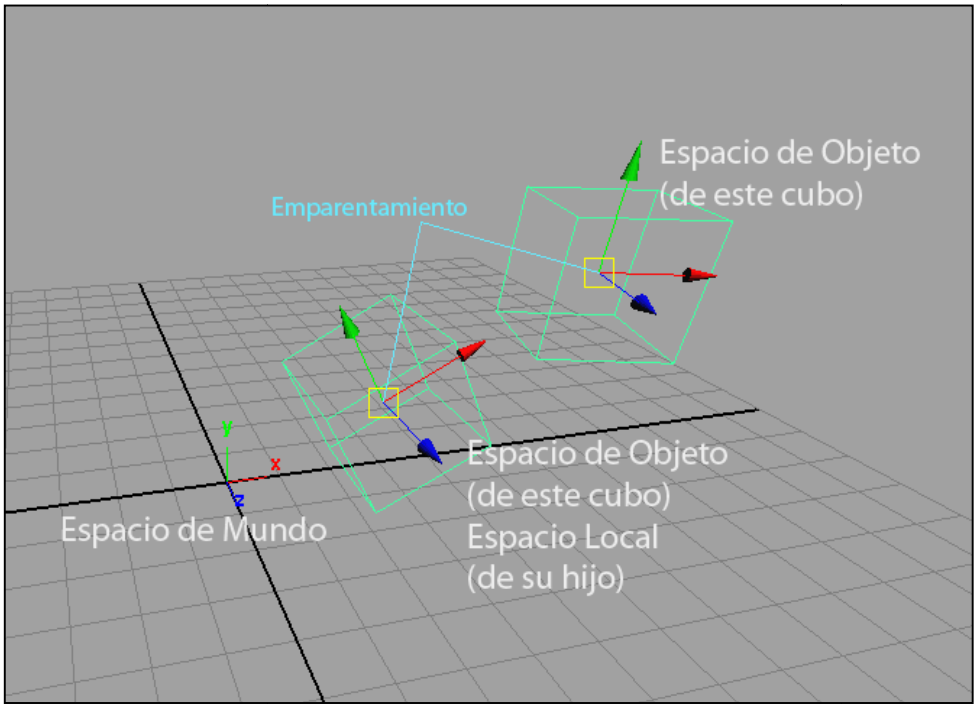


Imagen 30 : Diferentes espacios y sus relaciones.

Como puede verse en la imagen 30, el mundo está representado mediante una rejilla y su origen de coordenadas, 3 vectores: X unitario, Y unitario y Z unitario. En él vemos una jerarquía compuesta de dos cubos: cada uno de ellos también posee su propio origen de coordenadas.

El cubo que está más a la derecha tiene su posición y rotación expresadas en función del primero, es decir, está emparentado a él, por lo tanto el espacio del primer cubo es espacio de objeto para él y espacio local para su hijo. Su hijo tiene su propio espacio de objeto.

A.4 – TUBERÍA GRÁFICA

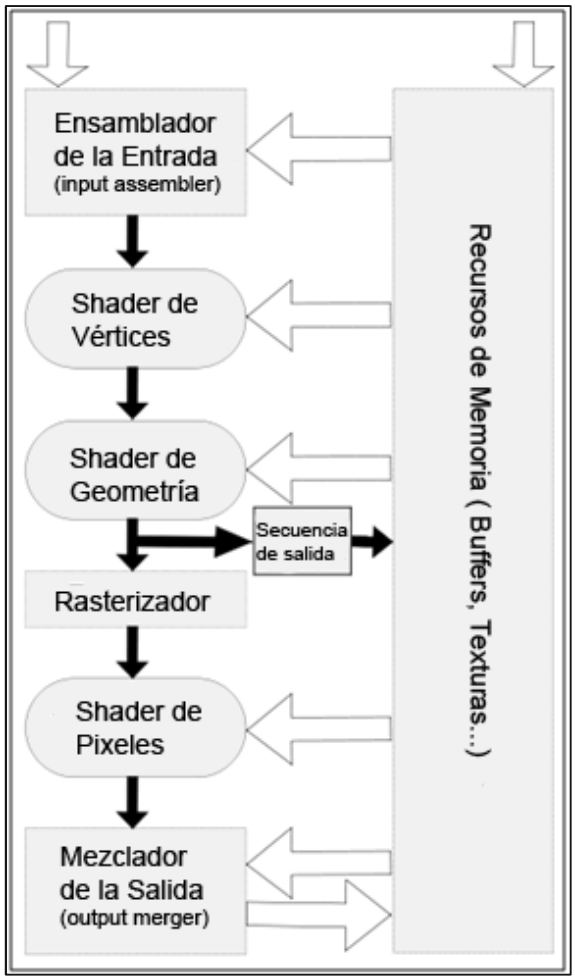


Imagen 31 : Etapas del hardware de clase DirectX10

Cuando se habla de gráficos acelerados por hardware un concepto importante es el de tubería gráfica. Se trata de una serie de manipulaciones que el hardware gráfico aplica de manera secuencial a las primitivas (vértices, triángulos) que le han sido enviadas por la aplicación, empleando cierta información (matrices de transformación, estados de renderizado, colores, etc.) también suministrada por ésta, con el fin de obtener la imagen final que ha de mostrarse en

pantalla. Una analogía con el mundo real sería la de una cadena de montaje, donde los materiales vírgenes pasan por varias etapas de procesado hasta llegar a producirse el resultado final.

Durante bastante tiempo la tubería (o "pipeline") consistió en su totalidad en etapas implementadas por completo en hardware, ofreciendo escaso margen de alteración por parte del programador. En los últimos años se ha ido introduciendo el concepto de tubería programable, gracias a la cual el programador de cada aplicación puede especificar mediante pequeños programas denominados sombreadores o "shaders" cómo se lleva a cabo el trabajo de cada etapa de la tubería. En la imagen 31 vemos un diagrama del pipeline empleado por el hardware de clase DirectX10.

Las únicas tres etapas no programables son:

- Ensamblador de la entrada: lee los vértices suministrados por la aplicación en un búffer con un formato determinado y los pasa al resto de la tubería.
- Rasterizador: Convierte los polígonos definidos por la lista de vértices en píxeles (proceso denominado rasterización) recortando además, aquellos polígonos o fragmentos de polígono que queden fuera del área visible de pantalla.
- Mezclador de salida: Mezcla distintos tipos de información (píxeles, estados de alpha, zbuffer, búffer de plantilla) y genera el resultado final.

Las etapas programables en este pipeline lo son mediante los siguientes shaders:

- Shaders de Vértices (Vertex Shaders): permiten realizar transformaciones sobre los vértices enviados a la GPU[24].
- Shaders de Geometría (Geometry shaders): de reciente introducción, permiten crear primitivas (vértices y triángulos) desde la GPU.
- Shaders de Pixeles (Pixel shaders): operan sobre "fragmentos", o píxeles aún sin colorear, y escriben sobre un búffer de sólo escritura.

De algunos de estos shaders se hablará más adelante y se expondrán ejemplos extraídos directamente de MAG.

Un último tipo de shaders ha sido introducido por DirectX11 muy recientemente: los compute shaders, que operan sobre hilos de ejecución, y escriben su resultado sobre buffers de memoria de lectura/escritura. De esta manera, permiten la ejecución de código de propósito general sobre la

[24] GPU (Graphical Processing Unit) o Unidad de Procesamiento Gráfico, es un procesador dedicado única y exclusivamente al procesamiento de gráficos para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos y o aplicaciones 3D interactivas. De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la CPU (procesador central del computador) del ordenador puede dedicarse a otras tareas para las que es más eficiente. Son las tarjetas gráficas de los computadores las que disponen d estos procesadores especializados.

GPU, de manera similar a la tecnología CUDA[25] de Nvidia, aprovechando las ventajas ofrecidas por el procesamiento en paralelo.

Existen varios lenguajes de alto nivel en que pueden ser escritos los shaders, principalmente HLSL (DirectX) y GLSL (OpenGL). Hasta hace unos años, la única alternativa posible era ensamblador, pero los fabricantes de hardware crearon especificaciones y compiladores para estos lenguajes (similares a C) facilitando así la tarea.

[25] **CUDA** son las siglas de (***C**ompute **U**nified **D**evice **A**rchitecture*) que hace referencia tanto a un compilador como un conjunto de herramientas de desarrollo creadas por la empresa nVidia que permiten a los programadores usar una variación del lenguaje de programación C para codificar algoritmos en GPU (nota 3) de nVidia.

APÉNDICE B – TECNOLOGÍAS UTILIZADAS.

A continuación explicaremos las tecnologías utilizadas durante el proyecto, Collada y ColladaDOM, y también detallaremos por encima su funcionamiento general dando unos cuantos ejemplos.

B.1 - COLLADA

Collada es un formato pensado para servir como "contenedor" de intercambio de datos entre aplicaciones que manejen información acerca de una escena virtual. Es complejo, pero flexible y da cabida a datos extra (normalmente información exclusiva de cada aplicación) que necesiten ser almacenados junto con la definición de la escena. Su creación fue fruto de un intento por establecer un estándar de intercambio de datos 3D en un mundo donde impera el caos de formatos.

Originariamente, Collada fue especificado por Sony Computer Entertaiment (SCE), división de videojuegos de Sony. Actualmente está siendo desarrollado y mantenido por Khronos Group que mantiene el copyright con Sony. Khronos Group es un consorcio de múltiples compañías entre las cuales se cuenta AMD/ATI, Apple, Google, Nokia, Nvidia y demás empresas de gran importancia. Su uso es libre y gratuito.

Realmente, Collada no es más que un esquema XML. Este esquema se puede consultar en la página oficial de Collada[26] (versión 1.4.1). Además, se puede descargar una versión la especificación completa del formato.

Hemos elegido Collada como formato de exportación e importación debido a varias razones:

- Es compatible con gran cantidad de software de desarrollo gráfico, y es probable que cada vez sea mejor soportado.
- Al estar basado en XML, es un formato ASCII y puede ser leído por un humano, lo cual facilita la depuración de exportadores/importadores.
- Existe bastante documentación en línea y es capaz de almacenar todos los datos necesarios para MAG.

Para su uso, nos hemos decantado por la versión 1.4 de la especificación, que aunque no es la más reciente, es la más consolidada. Además, no se ha hecho uso de las nuevas características que introduce la 1.5. En cualquier caso, Khronos Group asegura que ambas especificaciones son compatibles, por lo que en principio no debería haber problema al trabajar con el software más reciente.

---

[26] ver apartado "Enlaces Útiles"

La lectura y escritura de ficheros Collada es una parte importante de este proyecto, pues uno de los objetivos era construir sobre a un formato vivo en el mercado, en lugar de reinventar la rueda. Las rutinas de escritura/lectura de archivos en formato Collada escritas para el proyecto se limitan a implementar un subconjunto de la especificación del formato, debido a la extensión y complejidad del mismo y a la escasa necesidad de dar soporte a la funcionalidad ofrecida por la especificación completa.

Esta especificación abarca gran cantidad de elementos que forman parte de una escena tridimensional: mallas geométricas, texturas, animaciones esqueletales, sistemas de partículas, física, iluminación, cámaras, shaders, etc. A continuación detallaremos el formato de almacenamiento de los recursos relevantes para este proyecto: mallas, esqueletos, texturas y animaciones.

Nótese que como todos los XML, se compone de distintos elementos y son diferenciados mediante su nombre. Por ejemplo, el tag <XXX> indica la apertura de un elemento con “XXX” como nombre. Cada elemento además necesita “cerrarse”, y para ello se usa el tag </XXX>. Es posible abrir y cerrar un elemento en un único tag <XXX/>.

Además, cada elemento puede contener atributos, que sirven para definir propiedades de ese elemento. En este caso, el elemento <XXX tipo=”Nodo”> contiene un atributo llamado “tipo”, y especifica que su valor es “Nodo”.

Los elementos están distribuidos de manera jerárquica, pudiéndose entender un archivo .dae (y cualquier XML) como un árbol. Por ejemplo, en <XXX> <YYY/></XXX>, se dice que YYY es hijo de XXX. En principio, en un XML sin ningún esquema, un elemento puede tener cualquier tipo de atributos y un número potencialmente infinito de elementos hijos. Sin embargo, un esquema de XML (como es Collada) obliga a la existencia de unos determinados elementos con restricciones de qué y cuántos elementos hijos tener, así como el tipo de atributos pueden contener. La primera limitación de este tipo encontrada es que el elemento raíz de un fichero Collada debe ser <Collada>.

B.1.1 - GEOMETRÍA

El primer recurso gráfico contenido en un .dae que nos es de utilidad es la geometría.

La geometría es almacenada en elementos <geometry>, que se encuentran en <Collada><library_geometries><geometry>. Sólo necesitamos una geometría por personaje, así que tomaremos el primer elemento <geometry> aparezca en el archivo. Suele tener dos atributos llamados “nombre” e “ID”, pero su uso es opcional y aquí no van a ser de utilidad alguna.

Dentro de este elemento, se encuentra el elemento <mesh>, que define una malla poligonal, que es el tipo de representación de superficies que emplearemos aquí (Collada admite más tipos de representaciones, como NURBS por ejemplo). <mesh> tiene varios hijos, puede tener varios <sources>, un <vertices> y un elemento de definición de una geometría primitiva, tal como <lines>, <polylists> o <polygons>. Nosotros vamos a usar <triangles>, ya que es la primitiva más común y la más básica, asegurándonos la compatibilidad con la mayoría de programas de diseño 3D.

En Collada un elemento <source> es el usado para declarar una fuente de datos. Por ejemplo, para definir datos de coordenadas, un source simplificado sería:

```
<source id=" VERTEX_1" name=" VERTEX_1">
        <float_array count="24">
                -50 50 50 50 50 50 -50 -50 50 50 -50 50
                -50 50 -50 50 50 -50 -50 -50 -50 50 -50 -50
        </float_array>
        <technique_common>
                <accessor count="8" stride="3">
                        <param name="X" type="float"/>
                        <param name="Y" type="float"/>
                        <param name="Z" type="float"/>
                </accessor>
        </technique_common>
</source>
```

Como puede verse, el elemento <float_array> contiene un array de datos. El <technique_common> define cómo se deben leer estos datol. Es este caso, especifica que los datos del array se deben separar en conjuntos de 3 (stride = 3) y que el primero es una coordenada X, el segundo Y y el tercero Z.

El elemento <triangles> contiene uno o más elementos <input> y un <p>. Cada <input> hace referencia a un <source> que ya se ha definido antes. Para entenderlo mejor veamos un ejemplo:

```
<triangles count=”2”>
        <input offset="0" semantic="VERTEX" source="#VERTEX_1" set="0"/>
        <input offset="1" semantic="NORMAL" source="#NORMAL_1" set="0"/>
        <p>     0 0     1 3     2 1    0 0     2 1     3 2    </p>
</triangles>
```

El atributo count de <triangles> especifica la cantidad de triángulos definida en su interior.

Dentro de <triangles>, cada elemento <input> especifica un tipo de datos requerido por los vértices que forman los triángulos. En el ejemplo, vemos que los vértices necesitan un input “VERTEX”, que se trata de la posición del vértice, y un “NORMAL” que es por supuesto el vector normal del vértice. Hay un atributo, denominado “source”, que establece un enlace (comenzado por el carácter ‘#’) a estas estructuras de datos, definidas anteriormente como hemos visto. De esta manera para acceder a la lista de posiciones del elemento con atributo “VERTEX”, basta con seguir el enlace a un elemento que tendrá como identificador “VERTEX_1”. Este mecanismo de enlaces o referencias a otros elementos del XML es empleado a menudo en el formato.

El elemento <p> contiene un array de índices sobre los arrays referenciados por los elementos <input>. ¿Cómo saber a qué array corresponde cada índice y qué tipo de datos esperar? Cada <input> posee también un atributo “offset” que indica qué índices del array <p> hacen referencia a sus datos. En el ejemplo: 2 triángulos de 3 vértices, donde cada vértice se define con dos índices. El primer índice hace referencia a su posición (offset = 0) y el segundo a su normal (offset = 1).

Este es un ejemplo sencillo, pero se pueden almacenar muchos más datos por cada vértice, como la tangente, las coordenadas de textura, la binormal, etc.

B.1.2 - ESQUELETO

El formato en que está almacenado un esqueleto en Collada es algo más complejo que el anterior. La información necesaria para crear el esqueleto y pesar la geometría está dividida, por un lado la información de skinning y por otra la definición de las juntas y huesos.

La parte de skinning se encuentra en <Collada><library_controllers><controller><skin>. Cada <controller> tiene un nombre y un id, para poder referenciarlo desde otros lugares mediante los enlaces comenzados por '#' que hemos visto antes. Cada <skin> posee un atributo "source" que referencia la geometría sobre la que se debe aplicar[27].

<skin> contiene diversos elementos:

- <bind_shape_matrix>: 16 floats para definir la matriz de bind[28].
- Al menos 2 <source>:
  - Nombres: Uno contendrá los nombres (y por lo tanto el número) de cada hueso.
  - Pesos: Contiene todos los floats que serán los pesos de cada influencia entre un vértice y un hueso.
- <vertex_weights>:
  - Debe declarar dos <input>, uno que haga referencia al <source> de nombres, y otro al <source> con los pesos.
  - Incluye un <vcount>, en el que por cada vértice indica cuántos huesos le influencian.
  - Un elemento <v>, en el que por cada influencia (es decir, 0 o más por vértices, dependiendo del <vcount>) especifica dos índices, el primero hacia que hueso es la influencia y el segundo el peso de esa influencia en el <source> de pesos.

Veamos un ejemplo sencillo:

```
<skin >
        <source id="joints">
                <Name_array id="nombres" count="3">hue0 hue1 hue3</Name_array>
        </source>
        <source id ="weights">
                <float_array id="pesos " count="5"> 1.0 0.5 0.3 0.2 0.1 </float_array>
        </source>
        <vertex_weights count="4">
                <input semantic="JOINT" source="#joints"/>
                <input semantic="WEIGHT" source="# weights"/>
                <vcount> 3 1 2 1</vcount>
                <v>
```

[27] Collada da soporte a múltiples skins y geometrías por fichero, pero por lo general suelen haber una de cada.

[28] Ver la sección 6.1.1: Hopp, subsección Animación Esqueletal.

```
                    0 1   1 0   2 3   2 4
                    1 2   0 5   1 3   2 4
                </v>
            </vertex_weights>
        </skin>
```

En este ejemplo, se definen tres huesos con nombres hue0, hue1 y hue3. Como vemos en <skin> no se define la posición ni la rotación de los huesos, únicamente la información de pesado. En el segundo <source> se define un array con floats, que serán los pesos.

Finalmente, en <vertex_weights> se definen las influencias. Se referencian cada <source> usando la palabras "JOINT" o "WEIGHT" para indicar cuál es cada (los id's de los <source> no tienen ningún significado). En <vcount> se establece que el primer vértice estará influenciado por tres huesos, el segundo y el cuarto por un hueso y el tercero por dos huesos.

Concretamente, en <v> se puede ver que el primer vértice está influenciado por el hueso número 0 (hue0) con peso índice 1 (0.5), también por el hueso 1 (hue1) con peso 0 (1.0) y por el hueso 2 con peso 3.

En resumen, la estructura de los datos es similar a la de la geometría, detallada con anterioridad: elementos <source> que contienen los datos reales, y elementos <input> que los referencian mediante un array de índices.

La segunda parte para poder leer un esqueleto de un fichero Collada es más sencilla. En ella se definen la posición de cada hueso y su relación en forma de árbol con los demás. Ésta se define en <Collada><library_visual_scenes><visual_scene>. Al igual que muchos otros elementos, cada <visual_scene> puede contener un nombre y un id como identificadores, pero únicamente emplearemos el primero en aparecer.

Dentro del <visual_scene> encontraremos elementos <node>, los cuales almacenan la información relativa a las juntas del esqueleto. Cada nodo tiene varios atributos:

- Un nombre y un id, los cuales se utilizarán para referenciarlos.
- Un atributo "type", que puede tomar como valores "JOINT" o "NODE".
- En caso que sea de tipo "JOINT", además incluirá un atributo "sid" que tendrá como valor el nombre del hueso que se ha definido anteriormente en el array de nombres dentro de <skin>, estableciendo así una relación.

Los <node> de tipo "JOINT" por lo general tendrán elementos <node> hijos, estableciendo implícitamente la jerarquía de juntas del esqueleto. Además, tendrán varios elementos <rotate> y/o <translate> para establecer su posición, su giro o su orientación en el espacio tridimensional.

El elemento <node> de tipo "NODE" contendrá un elemento hijo <instance_controller>, y éste a su vez un elemento <skeleton>, que como valor tendrá el id del <node> padre del esqueleto. Es decir, establecerá cuál será el <node> raíz del esqueleto. Esto es útil para poder tener varios esqueletos con un mismo conjunto de huesos, solo que cambiando el hueso raíz (es decir, tener un subconjunto), pero prácticamente no se utiliza.

Veamos un ejemplo:

```
<visual_scene id="ejemplo" name=" ejemplo ">
        <node id="joint1" name="joint1" sid="hue0" type="JOINT">
                <translate sid="translate">0.134796 -0.016849 0</translate>
                <rotate sid="jointOrientZ">0 0 1 90</rotate>
                <rotate sid="rotateY">0 1 0 36.9126</rotate>
                <node id="joint2" name="joint2" sid="hue1" type="JOINT">
                        <translate sid="translate">5.45926 0 0</translate>
                </node>
        </node>
        <node id="nombrejemplo" name="nombrejemplo" type="NODE">
                <instance_controller url="#piel">
                        <skeleton>#joint1</skeleton>
                </instance_controller>
        </node>
</visual_scene>
```

La jerarquía de nodos empieza con el <node> llamado “joint1”, que hace referencia al hueso llamado “hue0”, visto en la primera parte, en el elemento <skin>. Los elementos hijos son traslaciones, rotaciones y orientaciones con sus parámetros correspondientes. Además, tiene un hijo <node>, que puede contener transformaciones adicionales. Las rotaciones y orientaciones pueden resultar confusas: están expresadas en términos de <eje, ángulo>, donde los tres primeros números definen la dirección del eje de giro y el último la magnitud del giro en grados.

Por otro lado, encontramos el otro <node>. El elemento <instance_controller> de su interior hace referencia mediante el atributo “url” al nombre del controlador en el cual se encuentra el elemento <skin> que corresponde al esqueleto de esta junta. Así identifica que información de skinning es necesaria para este esqueleto en caso que haya más de una, lo cual no es habitual pero el estándar lo permite. Además de eso, el elemento <skeleton> identifica cual es el <node> padre, que en este caso es el identificado como joint1.

B.2 - COLLADA DOM

DOM es un acrónimo de *Document Object Model* (Modelo Objetual del Documento), una manera de expresar la jerarquía de elementos de un formato tal como XML o HTML en términos de objetos de un lenguaje orientado a objetos.

Collada DOM es un **framework**[29] para desarrollo de aplicaciones que usen el formato Collada. Proporciona un API C++ mediante el cual se pueden cargar, consultar, modificar y escribir archivos Collada. En este proyecto ha sido empleado siempre que ha sido necesario trabajar con archivos

[29] Framework: Conjunto de capas y módulos software sobre el cual otro proyecto puede ser desarrollado y organizado.

.dae, tanto para abrirlos y cargar sus recursos en memoria como para escribir la información una vez alterada a disco.

Coincidiendo con la versión de la especificación de Collada, hemos elegido trabajar con la versión 1.4 de este framework.

Es un software de código abierto soportado por la misma organización que creó Collada, asegurando su compatibilidad y desarrollo. Hay otro software conocido para poder cargar Collada que es **FCollada**[30], escrito por "Feeling Sofware". Es también de código abierto y es usado por bastantes productos. Sin embargo, se ha abandonado su desarrollo desde el 2007, y por temor a errores o incompatibilidades, nos decantamos por Collada DOM.

[30] Ver sección "Enlaces Útiles"

APÉNDICE C – DEPENDENCIAS DEL PROYECTO

En este apéndice describimos brévemente las dependencias que tiene cada una de las piezas de nuestro proyecto, referencias a ellas y una pequeña explicación de cada una. La intención de ello es servir de referencia y consulta para gente interesada en realizar un proyecto parecido al nuestro.

C.1 - LIBRERÍA MAGINPUT

Depende de las librerías Minizip y TinyXML.

Minizip es una librería que se emplea para comprimir y descomprimir paquetes de archivos en formato .zip y .gz. Funciona bajo Linux y Windows. No soporta encriptación, compresión multivolúmen (span) y compresión por metodos antiguos.
Para usarlo hay que descargar la librería Zlib que permite crear gzip (.gz).

TinyXML es un pequeño y simple parseador de archivos XML (archivos de metadatos), que se puede integrar fácilmente en otros programas. Con el archivo XML genera un Document Object Model (un arbol de jerarquía) que puede ser leído, escrito y modificado fácilmente.

C.2 - LIBRERÍA MAGOUTPUT

Depende de Minizip, TinyXML y, además, de ColladaDOM, y DevIL.

ColladaDom A su vez depende de nuevo de: Minizip, TinyXML y además de Boost, LibXML2 y PCRE.

Boost es una conocida colección de librerías de código abierto (Open Source) que extienden la funcionalidad básica de C++.

LibXML2 es un parser escrito en C originalmente orientado al Gnome Project que también sirve para analizar archivos XML.

PCRE es un conjunto de funciones para comparación de expresiones regulares con la misma semántica y sintaxis que Perl 5.

DevIL son las siglas de Developer Image Library, y contiene funciones que han sido empleadas en nuestras librerías para exportar desde OpenGL los mapas de textura una vez mezclados haciendo uso de la aceleración por hardware que nos ofrece la GPU.

C.3 – MAGTOOLS: EL EDITOR

Depende de las librerías Qt, MAGInput, MAGOutput, y además, de Framework Hopp.

QT[31] es una biblioteca multiplataforma para desarrollar interfaces gráficas de usuario y también para el desarrollo de programas sin interfaz gráfica como herramientas de la consola y servidores.. Es producido por la división de software Qt de Nokia desde 2008.

Qt es utilizada en KDE, un entorno de escritorio para sistemas como GNU/Linux o FreeBSD, entre otros y utiliza el lenguaje de programación C++ de forma nativa, extendiéndolo mediante un sistema de metaobjetos.

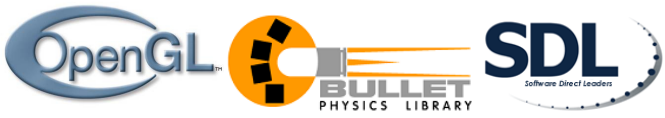
Funciona en todas las principales plataformas, y tiene un amplio apoyo en la industria. El API de la biblioteca cuenta con métodos para acceder a bases de datos mediante SQL, así como uso de XML, gestión de hilos, soporte de red, una API multiplataforma unificada para la manipulación de archivos y una multitud de otros para el manejo de ficheros, además de estructuras de datos tradicionales.

Distribuida bajo los términos de GNU Lesser General Public License (y otras), Qt es software libre y de código abierto.

[31] Ver “Enlaces Útiles”

Framework HOPP es un sencillo motor gráfico realizado por uno de los miembros del equipo de desarrollo del proyecto aquí expuesto, con anterioridad al proyecto en sí. Está construido en C++. Sus funcionalidades son, entre otras, las siguientes:

- Renderiza gráficos en 2D y 3D.
- Organiza las aplicaciones en “estados”.
- Contiene un completo grafo de escena.
- Utiliza el lenguaje de shading de OpenGL (GLSL).
- Se puede modificar a voluntad el “pipeline” de renderizado y postproceso.
- Posee un sistema de organización de recursos gráficos.



Depende a su vez del motor de física Bullet Engine (motor de físicas), de SDL (Simple Directmedia Layer) y de OpenGL.

APÉNDICE D – HERRAMIENTAS SOFTWARE UTILIZADAS

En este apéndice explicamos brevemente los diferentes software que hemos utilizado para realizar el proyecto: entornos de desarrollo Code::Blocks y QTCreator; los gestores de proyecto online Project Kenai y XP-Dev y otras herramientas como ColladaLoader.

D.1 - ENTORNOS DE DESARROLLO

Durante el desarrollo del proyecto hemos utilizado diversas herramientas para desarrollar. Code::Blocks, QTCreator y otras pequeñas aplicaciones adicionales.

D.1.1 - CODE::BLOCKS

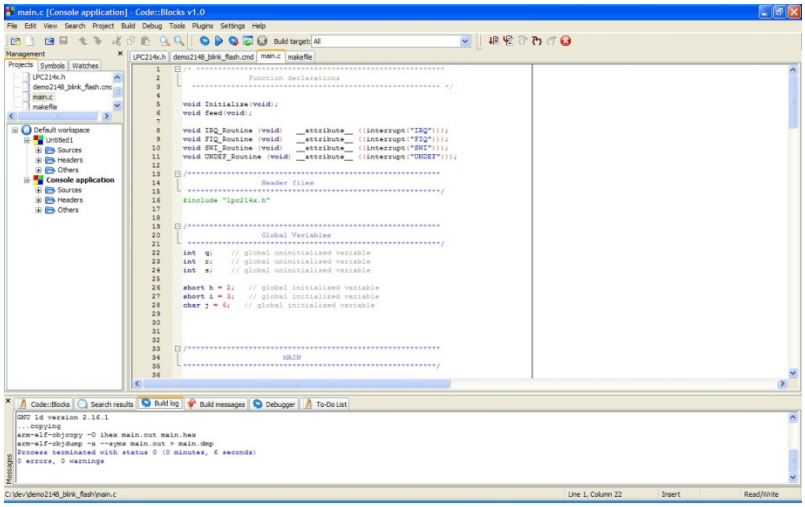


Imagen 32 : Captura de pantalla del IDE Code::Blocks.

Code::Blocks[32] es un entorno de desarrollo multiplataforma para desarrollar en C++. No contiene demasiadas novedades ni tiene un entorno muy amigable, pero es estable y proporciona todas las características que se le piden a un IDE[33], como autocompletado, compilación y ejecución desde el mismo entorno, búsqueda de usos, depuración, etc.

[32] Ver “Enlaces Útiles”

[33] Acrónimo de: Integrated Development Environment (Entorno de Desarrollo Integrado).

D.1.2 - QTCREATOR

El otro entorno utilizado fue QtCreator, utilizado para desarrollar el editor. Después de decidir que íbamos a utilizar QT para el desarrollo de la interfaz gráfica, usar la IDE oficial fue un paso lógico. Es un editor bastante nuevo, muy innovador y que ha ganado estabilidad y utilidad con los años. Se comporta muy bien a la hora de editar interfaces gráficas, pero parece que le faltan algunas opciones y extras que aumenten la comodidad cuando se maneja código.

El compilador usado en ambas ocasiones es el completo y conocido GNU Compiler Collection (GCC). Como hemos desarrollado el proyecto en Windows, hemos usado Mingw, que no es más que las herramientas de GNU portadas para la API Win32. Una de las opciones de descarga de Code::Blocks incluye MingW.

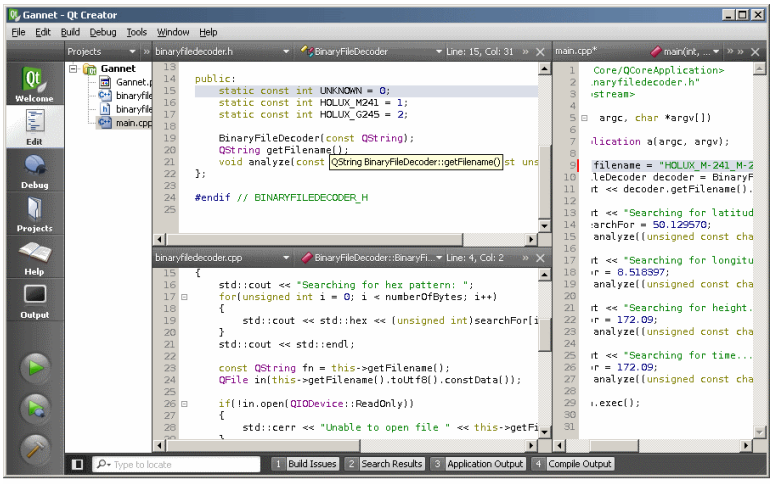


Imagen 33 : Captura de pantalla del IDE QTCreator

D.2 - GESTORES DE PROYECTO.

Para realizar el proyecto hemos utilizado un repositorio Subversion, para poder facilitar la tarea de trabajar en paralelo. Los integrantes del grupo ya estamos familiarizados con esta herramienta, y entre todos hemos probado bastantes servicios, incluyendo Sourceforge y Google Code.

D.2.1 - KENAI

Project Kenai beta

Project Kenai es un servicio de host para proyectos, con numerosas herramientas para poder gestionarlo. Fue creado por Sun Microsystems y lanzado en Septiembre de 2008. Aunque no era excesivamente estable y muchas veces se comportaba excesivamente lento, incluye muchos servicios que no tienen sus rivales, como SourceForge. A parte de poder establecer más licencias a tu código y de tener más opciones sociales para establecer contactos con otros desarrolladores, Kenai no solo da soporte a Subversion, sino también al mejor y más nuevo Mercurial. Posee tracking de bugs con JIRA o Bugzilla, una wiki, listas de emails, foro, etc. Es decir, un conjunto de servicios novedoso y actualizado, todo ello gratuito y sin publicidad.

Sin embargo, desde la compra de Sun Microsystems por parte de Oracle, van a cerrarlo e integrarlo con java.net. Por ello, aunque aún funciona correctamente, tuvimos que cambiar de host a mitad del proyecto.

D.2.2 - XP-DEV

XP-Dev es un host de proyectos. En su versión gratuita permite un repositorio Subversion privado con capacidad de 200 megas, además de herramientas desarrolladas por ellos de bug-tracking, wiki, un blog y foros. No tenemos ningún comentario negativo, ya que todos los integrantes estaban familiarizados con él. El repositorio Subversion es rápido y se comporta bien, además sus otros servicios funcionan correctamente.

Sin embargo, y aunque ofrecen planes profesionales, sus servicios parecen bastante básicos y falto de opciones, y creemos que no es el sitio adecuado para un proyecto que haga un uso extenso de estas herramientas y en el que intervenga mucha gente.

D.3 - APLICACIONES AUXILIARES

D.3.1 - COLLADA LOADER

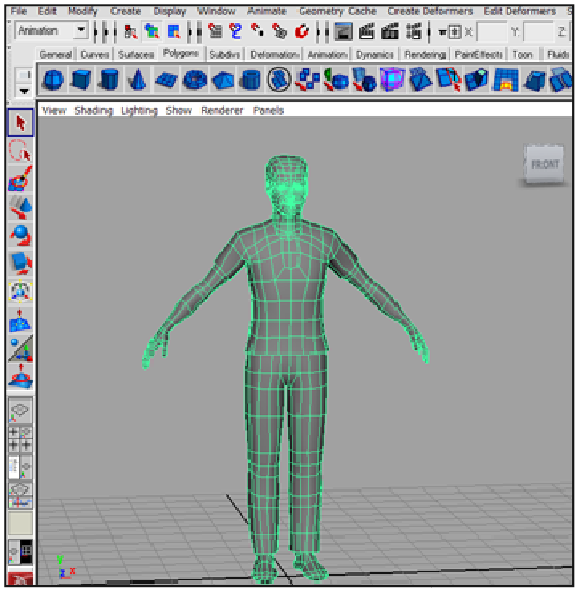
Un programa libre muy útil para poder ver el contenido de un Collada, parecido a nuestra ventana de visualización del editor MAGTools, pero sin posibilidad de modificarlo. Sirvió para etapas iniciales en las que no se sabía con exactitud qué datos contenían los .dae generados por otros programas de diseño, y para comprobar que los collada generados eran correctos y podrían verse.

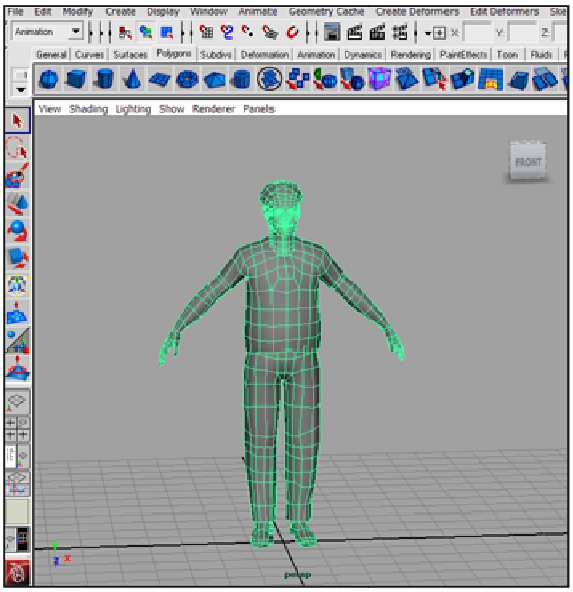
APÉNDICE E – UN EJEMPLO PRÁCTICO DE USO DE LA APLICACIÓN

Vamos a ilustrar el funcionamiento de la aplicación con un ejemplo/demostración informal donde emplearemos un personaje desarrollado durante el proyecto para generar un paquete .mag con un único personaje variación, que es el mismo personaje, pero de complexión más gruesa.

Este apartado está orientado para usuarios con un mínimo de conocimientos de grafismo 3D, ya que abarca el proceso de creación de recursos para el paquete, además de su utilización, con lo que no explicamos la manera de realizar los recursos, que excede el interés de esta sección.

Lo primero que es necesario es diseñar la malla base del personaje y desplegar sus coordenadas UV, explicadas en la sección 4.2. En nuestro caso obtenemos un humano de complexión media, utilizando para ello un conocido paquete de modelado y animación. El resultado es el siguiente:

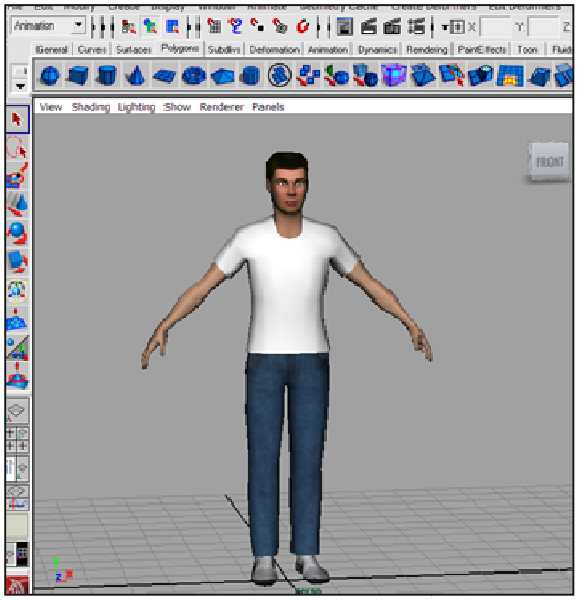


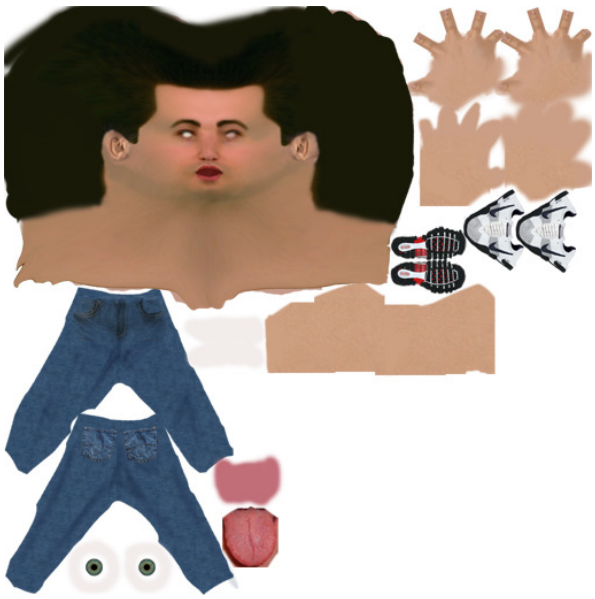


En este caso no realizamos animación, pero como es necesario que tenga al menos un hueso y una animación de ese hueso, hemos añadido un hueso en el pie, por ejemplo, lo asociamos a los vértices que lo rodean para que tenga relación con la malla y posteriormente realizamos una animación sin movimiento de un solo fotograma para que haga las veces de información de animación, aunque luego no haga nada.

Luego modificamos ligeramente el modelo para hacerlo un poco más grueso y lo guardamos exportándolo en formato Collada (.dae).

Después realizamos las texturas del modelo base, que luego servirán también para el grueso porque sus coordenadas de textura son idénticas.





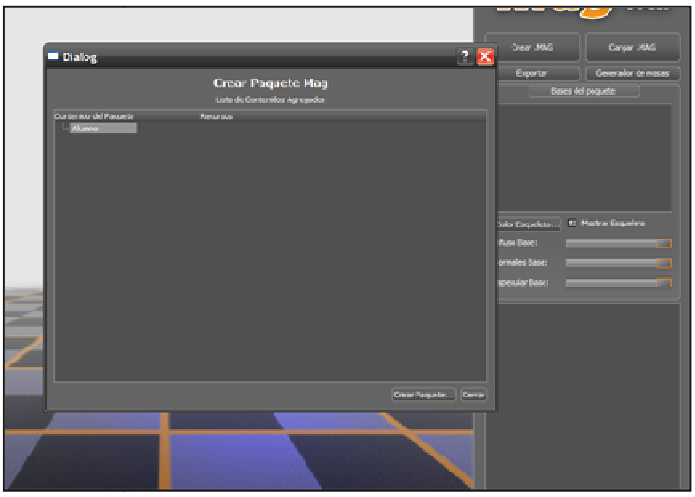
Aquí vemos el resultado de aplicar una textura de difuso (pintada en cualquier programa de edición de imagen digital) a nuestra malla base. También vamos a pintar una textura de overlays que nos permitirá sombreimprimir varios diseños en la camiseta y elegir sus colores. Para ello, tomamos una textura rgba y en cada uno de los 4 canales pintamos una imagen en escala de grises, empleando negro para las zonas que no deben verse y tonos de gris o blanco puro para el resto:

El primer canal tiene toda la parte de la camiseta en blanco (que es la esquina inferior derecha blanca que queda en el mapa de difuso superior, en color). Eso es porque el primer canal pinta toda la camiseta de un color. Después para el segundo canal dejamos solo las mangas y los bordes blancos, para que los pinte encima de otro color distinto y destaque, y los dos últimos canales son para incluir dibujos en la espalda y el pecho de la camiseta con un color diferente.

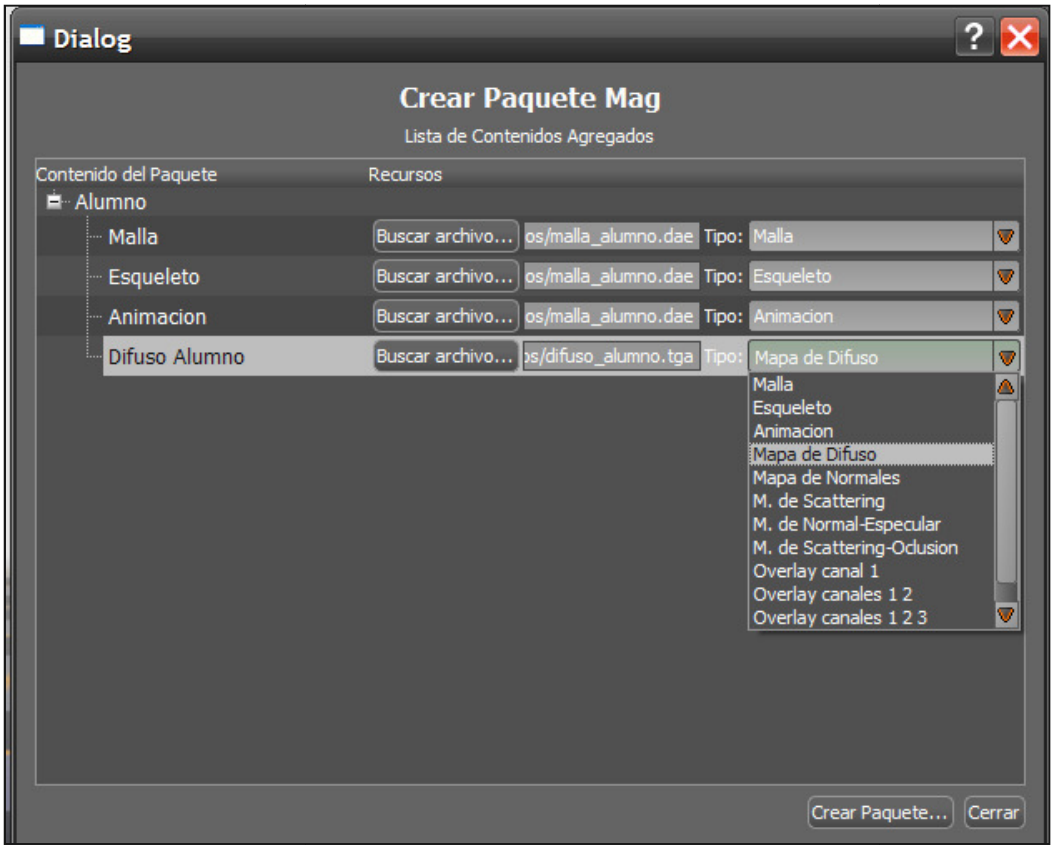
Esta textura de overlays es opcional, por supuesto, y servirá para darle variedad a los personajes.

Bien podría haberse hecho también para los pantalones, la piel o cualquier otro elemento, pintando de un gris más claro pero no negro (o blanco si queremos que sustituya el color por completo) la zona que corresponde a esas piezas del modelo de en la textura.

Posteriormente abrimos el editor MAG y pulsamos en “Crear .MAG”. Se nos abrirá una ventana nueva:

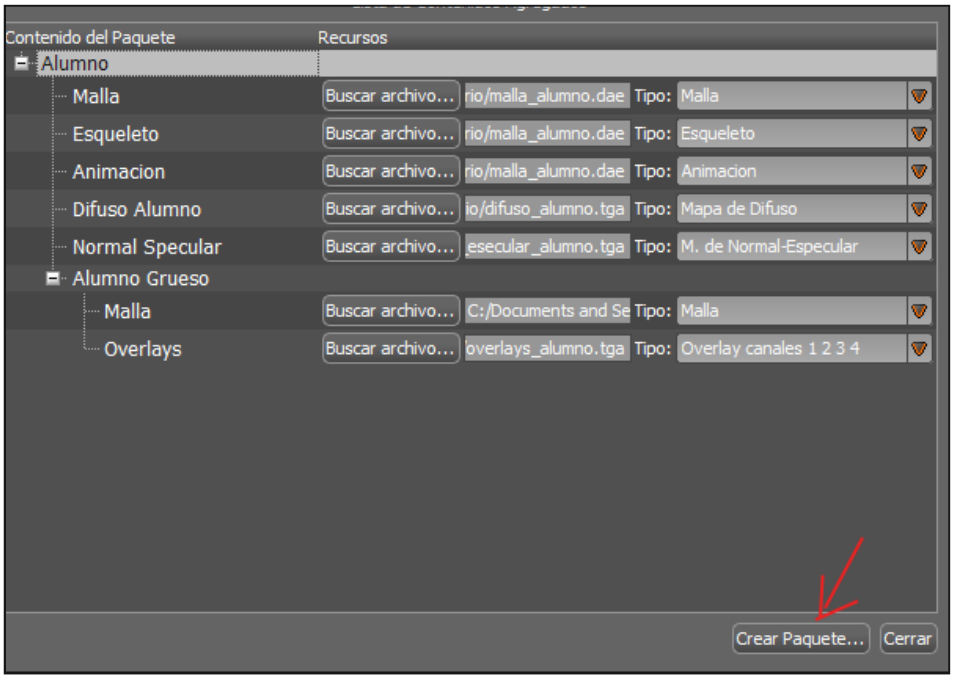


Haciendo click con el botón derecho del ratón, se nos permite añadir un nuevo personaje base al paquete, al que hemos dado el nombre “Alumno”. Si hacemos click secundario de nuevo sobre el personaje base recién añadido, es posible incluir en él un recurso gráfico nuevo o un personaje variación. Optamos por incluir 3 recursos extraídos del mismo archivo .dae que hemos exportado del software de diseño de nuestra elección: malla, esqueleto y animación. Además incluimos una textura de color difuso.



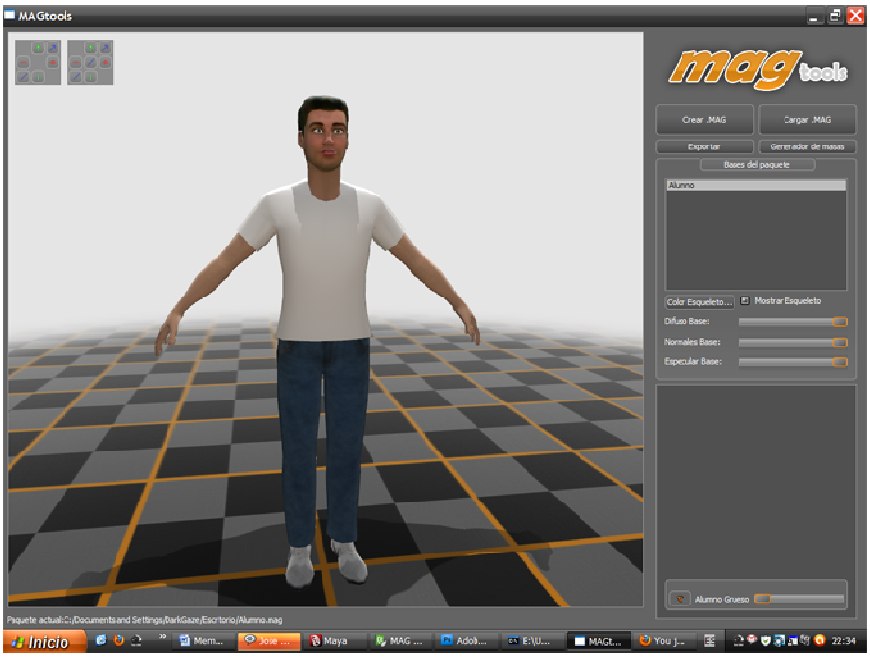
Ahora sí, añadimos una nueva variación a nuestro personaje. La llamamos “Alumno Grueso” e incluimos en ella los dos recursos que van a cambiar el aspecto base del personaje: la malla y una

textura de overlay de 4 canales. Para cada recurso que incluyamos, debemos especificar cómo se deben tratar los datos que contiene: es posible usar un mapa de normales como mapa de color difuso, aunque su utilidad es cuestionable.



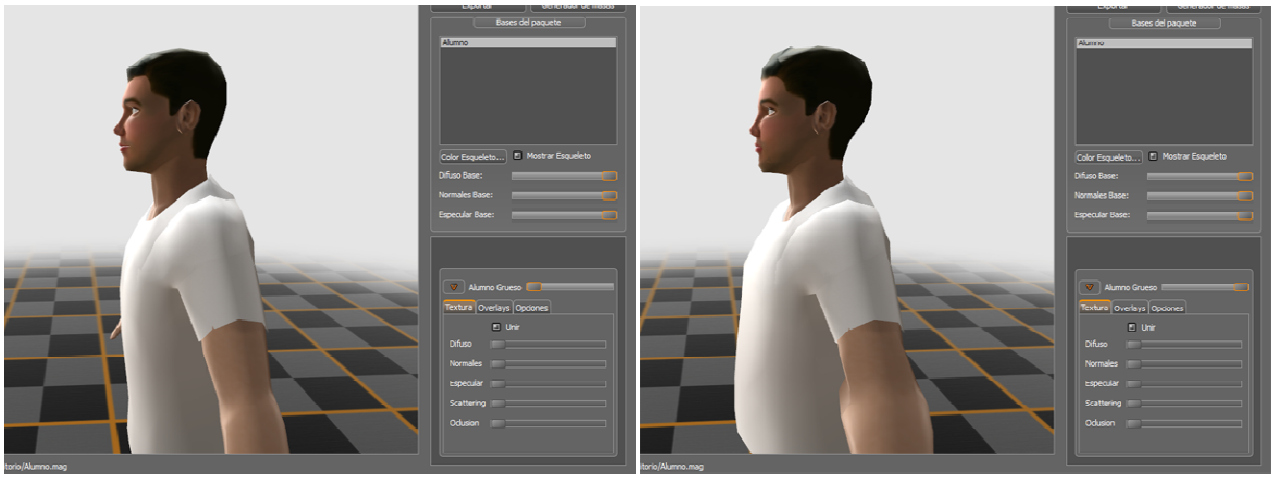
Una vez hemos terminado de incluir recursos en el paquete, es necesario guardarlo. Para ello haremos click sobre el botón “Crear Paquete...”, que nos abrirá una ventana de exploración para que elijamos el lugar donde se debe almacenar el paquete resultante.

Ya hemos creado un sencillo paquete, podemos abrirlo directamente desde la aplicación para empezar a utilizar su contenido para la generación de personajes. Pinchamos en “Cargar .MAG” y escogemos el paquete que acabamos de realizar. En la lista de personajes base del paquete, aparecerá “Alumno”. Haciendo click sobre él, tras unos segundos de carga, la base aparecerá en la vista 3D.

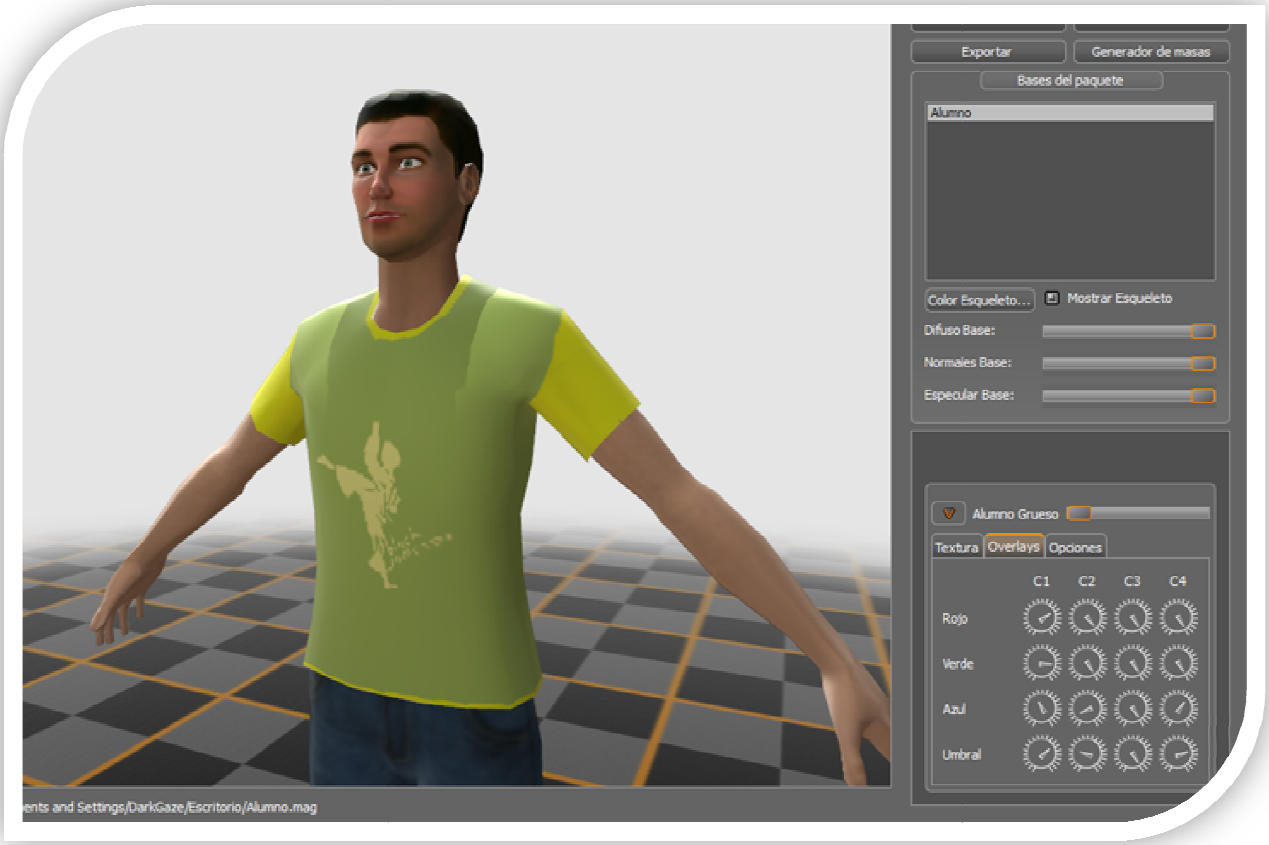


Además, en el listado de variaciones habrá aparecido un pequeño panel desplegable, que si abrimos nos mostrará lo que aparece en la siguiente imagen:

Comprobamos que, efectivamente, los deslizadores de Difuso, Normales, Especular, Scattering y Oclusión aparecen sombreados en gris ya que no hemos definido ningún recurso de estos tipos en la variación “Alumno Grueso”. Únicamente el mapa de overlays y el deslizador de mezcla superior.



Desplazando el deslizador principal de la variación, que es el único activo, vemos en tiempo real cómo cambia la complexión de nuestro personaje. No tenemos porqué emplear uno de los dos extremos (delgado o grueso), cualquier valor intermedio es calculado para nosotros por MAG.



Manipulando los manejadores de la capa de overlay, podemos estampar dos diseños al mismo tiempo en la camiseta (mangas y cuello e imagen del pecho) y cambiar el color de cada una y del fondo de la camiseta de manera independiente.

Y ya tenemos nuestro personaje, un término medio entre delgado y grueso, con una vestimenta de llamativos colores. Sólo hace falta pulsar en “Exportar” y elegir un lugar donde guardar un archivo

.dae con la malla del personaje y una textura de color difuso (overlays ya sobreimpresos en ella con los colores y patrones escogidos), listos para ser usados en cualquier aplicación.

En este ejemplo hemos visto cómo, a partir de un par de mallas y texturas (difuso y overlays) podemos generar personajes con múltiples indumentarias y con ligeras variaciones de complexión. Por supuesto, es posible añadir más variaciones: con overlays para el pantalón, tatuajes, color del pelo y piel, una malla para personajes altos y otra para personajes de pelo largo, y mapas de normales para pelo liso y rizado, se podría generar una gran cantidad de humanos de apariencia variopinta, activando varias variaciones con distinta intensidad y atributos cada una: altos, rubios y de pelo largo, de escasa estatura, gruesos con la piel morena y pelo ondulado o con tatuajes de varios colores por todo el cuerpo. Si además recordamos la posibilidad de incluir varias mallas base en el mismo paquete, podremos generar del mismo paquete ejecutivos elegantemente vestidos o jugadores de fútbol, cada uno con un aspecto distinto, y ver el resultado final al instante con una gran calidad gráfica.

Dada la genericidad de las librerías y del editor, no existe limitación alguna en cuanto al tipo de personajes que pueden formar parte de un paquete .mag.

Lo demostramos en la siguiente página.

Como demostración de lo que puede conseguirse alterando todos los datos de una variación con respecto a su base, incluimos aquí 4 etapas de la transformación continua de un cangrejo en una de sus variaciones, que sucede mientras se reproduce su animación esqueletal.

REQUISITOS TÉCNICOS

Los requisitos técnicos mínimos para el funcionamiento del proyecto son:

-512 Mb de RAM

-Tarjeta gráfica compatible con OpenGL 2.1 o superior.

-Microsoft Windows, recomendado a partir de la versión XP.

ENLACES ÚTILES

Esta sección aglutina enlaces web con información que hemos encontrado relevante durante el proceso de desarrollo del proyecto:

Productos parecidos al nuestro explicados en el apartado de Estado del Arte:

1. Quidam 3D: http://www.n-sided.com/3D/quidam.php?rub=1
2. Poser: http://poser.smithmicro.com/

Sobre Collada, lo mejor es consultar su página oficial, que es una wiki con mucho contenido:

3. https://collada.org/mediawiki/index.php
4. http://www.khronos.org/files/collada_schema_1_4
5. http://www.khronos.org/files/collada_spec_1_4.pdf

De COLLADADom tenemos el proyecto en:

6. http://sourceforge.net/projects/collada-dom/

Para la generación de la documentación y comentarios del código existe la página de Doxygen y sus ejemplos:

7. http://www.stack.nl/~dimitri/doxygen/

Acerca de FCollada:

8. http://www.collada.org/mediawiki/index.php/FCollada

Sobre transformaciones de matrices recomendamos la wikipedia y otras páginas:

9. http://en.wikipedia.org/wiki/Transformation_matrix
10. http://www.codeproject.com/KB/openGL/Space_Matrix.aspx

Sobre otros sistemas de representación de superficies y sobre gráficos 3D en general:

11. http://es.wikipedia.org/wiki/V%C3%B3xel
12. http://www.euclideanspace.com/maths/
13. http://www.gamedev.net

Sobre el bloqueo de gimbals, que es uno de los problemas de representar las rotaciones sobre un eje arbitrario como concatenación de tres rotaciones en tres ejes de referencia x,y,z (solucionable empleando cuaterniones) podemos referirnos a la wikipedia:

14. http://en.wikipedia.org/wiki/Gimbal_lock

Sobre cuaterniones recomendamos acudir a una subsección de un artículo de Wikipedia.

15. http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation#Qualitative_description_of_the_advantages_of_quaternions

Para las distintas aplicaciones utilizadas tenemos unos cuantos enlaces que pueden resulta de utilidad:

Página oficial del proyecto Kenai:

16. http://www.kenai.com

Información sobre Subversion:

17. http://en.wikipedia.org/wiki/Subversion_(software)

Información sobre Mercurial:

18. http://en.wikipedia.org/wiki/Mercurial_(software)

Página oficial de XP-Dev:

19. http://www.xp-dev.com/

Página oficial de GCC:

20. http://gcc.gnu.org/

Página oficial de MinGW:

21. http://www.mingw.org/

Página oficial de Code::Blocks:

22. http://www.codeblocks.org/

Página oficial de Qt:

23. http://qt.nokia.com/products/developer-tools/

LISTA DE IMÁGENES

LISTA DE ECUACIONES

BIBLIOGRAFÍA

[COL14] Documento de Especificación de COLLADA 1.4

[COLDOM] COLLADA DOM 1.4.1 Programming Guide

[COLDOC] Documentación de COLLADA DOM

[QTDOC] Documentación de QT

[SHR2007] Shreiner, D., *The OpenGL Programming Guide ("The Red Book")* 7ºed. Addison-Wesley Professional. 2007

[ROS2009] Rost, Randi J., *The OpenGL Shading Language ("The Orange Book")* 3ºed. -Wesley Professional . 2009

[GRE2007] Green, C., *Improved Alpha-Tested Magnification for Vector Textures and Special Effects.* Valve 2007.

[REE1987] Reeves, W. T., Salesin, D. H., Cook, R. L. *Rendering Antialiased Shadows with Depth Maps*. 1987.

[DON2006] Donnelly, W., Lauritzen, A. *Variance Shadow Maps*. 2006.

[COH1999] Cohen, J. *Appearance-presenving Simplification of Polygonal Models*. 1999

[EBE2006] Eberly, David H. *3D Game Engine Design: A Practical Approach To Real Time Computer Graphics*. The Morgan Kaufmann Series in Interactive 3D Technology. 2006.

[BUI1973] Bui, T. P. *Illumination of Computer Generated Images*. 1973

[BAR2004] Barrera, T., Hast, A., Bengtsson, E. *Incremental Spherical Linear Interpolation*. 2004

[WAL1981] A.Wallace, B. *Merging and Transformation of Raster Images For Cartoon Animation*. 1981

[MIT2007] Mitchell, J., Francke, M., Eng, D. *Illustrative Rendering in Team Fortress 2*. 2007

[GOR1971]Goraud, H., Computer Display of Curved Surfaces. 1971